1. Itse1359-1000-Preface
2. Itse1359-1010-Getting Started
3. Itse1359-1015-Program Organization
4. Itse1359-1020r-Review
5. Itse1359-1030r-Review
6. Itse1359-1050-Introduction to Scripts
7. Itse1359-1060-Syntax
8. Itse1359-1065-Visualizing Python
9. Itse1359-1080r-Review
10. Itse1359-1090-Lists Part 2
11. Itse1359-1110-Nested Tuples
12. Itse1359-1110r-Review
13. Itse1359-1120-Empty and Single-Item Tuples
14. Itse1359-1120r-Review
15. Itse1359-1130-Unpacking Tuples
16. Itse1359-1130r-Review
17. Itse1359-1210r-Review
18. Itse1359-1220-Operators
19. Itse1359-1220r-Review
20. Itse1359-1230r-Review
21. Itse1359-1240r-Review
22. Itse1359-1250-Nested Loops
23. Itse1359-1250r-Review
24. Itse1359-1260-Loop Modifiers
25. Itse1359-1280-Function Arguments
26. Itse1359-1410-Overview of Python classes
27. Itse1359-1410r-Review
28. Itse1359-1450-Inheritance
29. Itse1359-1610-GUI Programming
30. Itse1359-1720-Doctest
31. Itse1359-1720r-Review

32. Itse1359-1900-Preface to Networking and Databases
33. Itse1359-1930-Dbm and Shelve Databases
34. Itse1359-2110-Preface to Putting Python to Work
35. Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer

Itse1359-1000-Preface
This module is the preface for a collection of modules designed for teaching ITSE 1359 Introduction to Scripting Languages: Python at Austin Community College in Austin, TX.

Table of contents

- Welcome
- Information about the course
    - Course description
    - Prerequisites
    - Course rationale
- Learning resources
    - General
    - Primary learning resources
- How to use this instructional material
- An editorial comment
- Reinforcing what you are learning
- Miscellaneous

Welcome

Welcome to the course material for *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

(Note to blind and visually impaired students: most of the material for this course is presented in plain text format and should be accessible using an audio screen reader or a braille display. While there are a few bitmapped images that aren't accessible, that material will not be needed for assignments or included on tests.)

Information about the course

Course description

As of September 2014, the description for this course reads as follows:

> Introduction to scripting languages including basic data types, control structures, regular expressions, input/output, and textual analysis.

Prerequisites

One semester of programming or departmental approval.

Course rationale

This course is an introduction to scripting languages and Python. The purpose of the course is to prepare students for building scripts that control a sequence of program steps such as those used in developing testing and deploying software. A modern scripting language, Python, is used as an example of a scripting language.

Learning resources

General

I have long been a proponent of free and open educational resources for college students. This version of this course does not require students to purchase printed textbooks. Instead the learning resources for the course consist of a large number of online resources arranged in the **following major sections** :

1. Language Fundamentals - Demonstrate the basic techniques used to create scripts for automating system administrative tasks.

2. Testing - Design, code, and test Python applications using Python doctests and unit testing tools.
3. Text processing - Demonstrate the use of string operations and regular expressions in processing text.
4. Networking and Databases - Demonstrate the use of Python in developing applications using networking and databases.

Primary learning resources

You are probably reading this document online at cnx.org , otherwise known as **openstax cnx** . As of October 2014, two views of the document are available: an *openstax* view and a *legacy* view. The material is the same in both views. Only the format and the top-level organization is different.

In the *openstax* view, the group of documents is referred to as a **book** and each document is referred to as a **page** .

In the *legacy* view, the group of documents is referred to as a **collection** and each document is referred to as a **module** .

You should not think of this book or this collection as a comprehensive tutorial on Python programming. Instead, you should think of it as an *annotated guidebook* to free online resources that constitute the primary learning resources for the course.

In some cases, I will explain the material in detail and illustrate the material with working programs. In other cases, I will partially explain the material and refer you to one or more online resources for additional explanations. In some cases, I will simply refer you to one or more online resources for the complete explanation. In all cases, I will either attempt to explain the material or refer you to one or more online resources for a full or partial explanation.

For many of the modules, I will also provide a *review module* containing questions and answers to help you retain the important information provided by the module.

References to free online resources will include but will not be limited to the following websites listed in no particular order:

- How To Think Like a Computer Scientist - Interactive Version
- How to Think Like a Computer Scientist
- Skulpt
- Online code visualizer
- learnpython.org
- Python at Codecademy
- Program Arcade Games With Python And Pygame
- tutorialspoint -- Python Tutorial
- The Python Tutorial
- A Beginner's Python Tutorial
- Another Beginner's Python Tutorial
- Diving Into Python
- Python/C API Reference Manual
- The Python Language Reference
- The Python Standard Library
- Hands-on Python Tutorial
- learnpython.org
- Non-Programmer's Tutorial for Python 3
- Python3 Tutorial
- Testing Your Code -- The Hitchhiker's Guide to Python
- TkDocs
- Hello Tkinter
- The Tkinter Grid Geometry Manager
- Python Course
- Tkinter 8.5 reference: a GUI for Python
- The Invent with Python Bookshelf *(many online books, some free, some not free)*
- Python Scientific Lecture Notes

Several of the assignments require the use of Python Turtle Graphics. Here are links to some useful references on Turtle Graphics:

- 24.1. turtle - Turtle graphics - Python 3.4.3rc1 documentation
- Background Drawing Graphics - 50 Examples 1.0 documentation

- How to Think like a Computer Scientist Interactive Edition
- Notes on using Python's turtle built-in commands
- Simple drawing with turtle - Introduction to Programming with Python

How to use this instructional material

Everyone is welcome to use this material in any manner that is consistent with the Creative Commons license under which it is published. However, if you are enrolled in Prof. Baldwin's section of this course at Austin Community College in Austin, TX, there are some other things that you need to know.

Although the URL may change in the future, for the Spring 2015 semester, you will find the online syllabus for the course at http://www6.austincc.edu/directory/info.php?id=baldwin .

Also, although the URL may change in the future, for the Spring 2015 semester, you will find the entry to the college website for the course at http://www.austincc.edu/baldwin/ .

You will also have access to a Blackboard site that is specifically tailored to the course. You will find additional learning resources on that site including programming assignments and tests, assignment and test schedules, etc. The order of the programming assignments and tests mirrors the order of the topics covered in this Ebook. Therefore, you should study the material in the Ebook from start to finish in parallel with your efforts to complete the assignments and the tests.

An editorial comment

In the opinion of this author, the Python programming language in and of itself is not a particularly interesting programming language. In fact, to a programmer accustomed to compiled, strongly-typed programming languages such as C, C++, and Java, the Python programming language seems to be a little "sloppy" and fraught with pitfalls. However, in the grand scheme of things, as of 2015, Python is an **extremely important** programming language.

The importance of Python derives not from the language itself, but from the hundreds of independent open source Python libraries that have been developed by others in such areas as image manipulation, networking, plotting and graphics, engineering and scientific programming, web development, gaming, cryptography, database, geographic information systems *(GIS)* , audio and music, presentation, XML processing, etc. In fact, it is hard to come up with a programming application area where someone has not already supplemented the basic Python programming language with an open-source library designed for use in that application area. Many of those libraries are Python wrappers for compiled C code providing speed and efficiency not normally associated with interpreted languages such as Python. A long list of such library modules is provided at UsefulModules . Many of the links on that page point to other pages that also contain lists of links. An even longer list is provided on the SciPy Topical Software page.

Reinforcing what you are learning

Although not required by the course, I highly recommend that you also study the material in the section of this Ebook titled Putting Python to Work in addition to the required material described earlier . Seeing how Python is actually used in practice may help you to develop and retain the knowledge required to succeed in the course.

Similarly, I also recommend that you study the material on the following three websites in parallel with the material on this website to reinforce what you **are learning** :

1. Python at Codecademy
2. Program Arcade Games With Python And Pygame
3. How To Think Like a Computer Scientist - Interactive Version

The first website listed above provides an interactive online Python tutorial that will help you learn the fundamentals of Python programming and test your progress along the way.

The second website listed above will not only help you learn about Python programming, but may also provide you with some enjoyment along the way. This website provides both text and video instruction, along with quizzes and programming projects to teach you how to use the Python language to write interactive arcade games involving sound, graphics, etc. While much of that material is beyond the scope of this course, you may find that learning that material will make the course more enjoyable.

The third website listed above is an excellent free online interactive Python textbook designed specifically for use in courses commonly referred by computer science academics as CS-1 courses. The non-interactive version of the book is located here .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1000-Preface
- File: Itse1359-1000.htm
- Published: 10/13/2014
- Revised: 01/21/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1010-Getting Started
This module explains how to get started programming with Python.

Table of Contents

- Preface
  - What you will learn
    - Beginners
    - Programmers
    - Python programmers
  - Windows would be handy
  - Practice, practice, and more practice
  - Viewing tip
    - Figures
- Why use Python?
  - What is Python?
  - More facts, less hype
    - Learn OOP without the details
    - A beginner-friendly language
    - Sneaking up on OOP
    - Python has an interactive mode
    - Python has a non-interactive mode
    - Python combines interactive and non-interactive modes
    - Small core, large library
- Let's Write a Program
  - Getting ready
    - Download and install the software
    - Starting the programming environment

        - Enough talk -- let's begin
        - What do you see?
        - The python manuals
        - The library reference
        - IDLE (Python GUI)
        - What does it look like?
        - What is a GUI?
        - The Python (command line) selection
        - The command-line window

    - Your first (interactive) Python program

        - Hello World in Python
        - The Python prompt >>>
        - Let's do it
        - Congratulations
        - Python is ready for more

- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you will learn

There is something for just about everyone here. Beginners start at the beginning, and experienced programmers jump in further on. You simply need to enter the collection of modules at the point that best fits your prior programming knowledge.

Beginners

The official prerequisite for this course is *"One semester of programming or departmental approval."* However, if you don't know anything about programming, these modules will help you to overcome that deficiency by taking you from ground zero to object-oriented computer programming using Python.

Programmers

If you already know how to program in some other language, these modules will teach you how to program using the Python scripting language.

Python programmers

If you already know how to program using Python, you can probably skip the modules in the "Language Fundamentals" section.

Windows would be handy

Although not absolutely necessary, it would be very handy for you to have access to, and know how to use Microsoft Windows. The programming skills that you will learn are applicable to a broad range of operating systems. However, I will use Microsoft Windows as my teaching platform, so you will need to be able to follow instructions explained in Windows jargon.

Practice, practice, and more practice

I am going to provide you with a lot of information, but you will also need to put your brain in gear and write a lot of programs to retain the information. Very few people become expert musicians without a lot of practice. Similarly, very few people become expert programmers without a lot of practice.

So, if you want to learn to program, plan to spend a lot of time in front of your computer, not just reading, but programming as well.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

(Note to blind and visually impaired students: the Figures in this module are bitmap images and are probably not accessible using an audio screen reader or a braille display. However, those Figures are simply screen shots of material that you can create on your own screen, which is probably accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Python 3.4.1 Shell
- Figure 2 . Python (command line) window.
- Figure 3 . Python Hello World.

Why use Python ?

With so many programming environments available, why use Python?

First of all, it's free. I like that. You won't need to invest an arm and a leg just to get started. All you will need to do is go to http://www.python.org/ and download the software.

What is Python ?

You might also want to visit The Python Tutorial , which is where I extracted the following:

Note: What is Python ?
Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

More facts, less hype

So, beyond the hype, did I really choose Python because it is an interpreted, interactive, object-oriented programming language that is relatively easy to learn? The name of this course is *"ITSE 1359 - Introduction to Scripting Languages: Python"* so I didn't really have a choice.

Learn OOP without the details

With Python, you can learn some aspects of object-oriented programming without the need to understand many of the complex details. Note, however, that Python supports a very abbreviated form of OOP. You will have much more to learn about OOP when you move on to C#, C++, or Java.

A beginner-friendly language

Python is a beginner-friendly language that takes care of many of the complex details for you behind the scenes. This makes it possible for you to concentrate on the big picture of what your program is intended to accomplish without getting bogged down in detail.

Sneaking up on OOP

With Python, you can "sneak up" on object-oriented programming concepts. You don't have to deal with the object-oriented nature of the language on

the first day, or for a long time, for that matter. *(With Java, for example, you can't write and understand even the simplest program without encountering a need to understand some object-oriented programming concepts.)*

Python has an interactive mode

The interactive mode makes it easy to try new things without the *edit-compile-execute* cycle of other programming languages such as Java and C++. Interactive mode is a friendly environment for persons in the learning stages of a language.

Python has a non-interactive mode

Once you know that something works properly, it is very easy *to "put it in a can"* so to speak, and then execute it as a script. This lets you execute it without retyping it every time.

Python combines interactive and non-interactive modes

This combination gives you the ability to use previously written script files while working in the interactive mode. Using this approach, you are able to minimize your interactive typing effort while still writing and testing large Python programs in interactive mode.

Small core, large library

Like Java, Python has a small compact core and a large, extensible library . Thus, much of what you will need to do has already been written and tested for you. Your task will be to write the code to glue those library components together, and to write new capabilities on an as-needed basis.

Let's Write a Program

Getting ready

Download and install the software

The first step is to go to http://www.python.org/ with your web browser. Download, and install the Python software using the download link that you will find there.

I'm going to assume that either you already know how to do this, or you can get help from a friend. In other words, I'm not going to try to explain how to download and install the Python software.

> Note: If you are working in an ACC computer lab on the Northridge campus, you don't need to download and install the Python software. It has already been installed. If you are working in an ACC computer lab on a different campus and you don't find the Python software, ask the lab manager to download and install it. If you are working on your computer at home, at some point you will need to download and install the Python software.

Starting the programming environment

As I mentioned earlier, all of my instructions will be based on the use of Python running on my Windows operating system.

However, as I also mentioned earlier, you have many choices for using Python. If you are running on some other platform, you will need to translate my instructions from Windows jargon into the jargon of your platform.

Enough talk -- let's begin

Find the *Start* button on the task bar on your Windows desktop. Select

Start/All Programs/Python x.y

where x.y is the version of Python that you have downloaded and installed. At the time of this writing, I have version 3.4.1 installed on my computer, and the selection shows on the menu as Python 3.4.

What do you see?

When you make this selection, you should see a menu having at least the **following five options** :

- IDLE (Python GUI)
- Module Docs
- Python (command line)
- Python Manuals
- Uninstall Python

The Python Manuals

Don't be bashful about selecting and reading the Module Docs and the Python Manuals. There is a wealth of information contained there, including a tutorial.

The library reference

A good way to get a feel for the breadth and power of Python is to select **Python Manuals** from the menu and then select the **Library Reference** link. *(The Library Reference is also available online as of the time of this writing.)*

Most of what you see there probably won't mean much to you at this point in time, but hopefully will be familiar territory after you complete this

course.

IDLE (Python GUI)

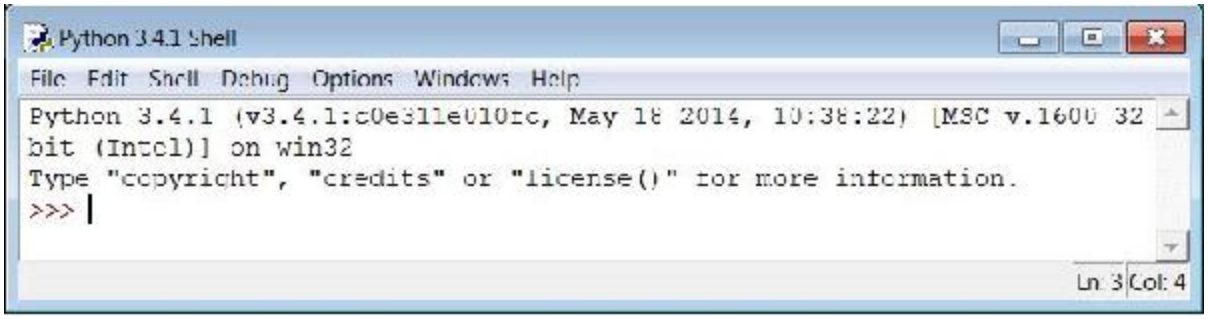
Selecting **IDLE (Python GUI)** from the menu brings up a window titled *"Python 3.4.1 Shell"* on my computer. This is one of the interactive programming environments available with Python on Microsoft Windows.

What does it look like ?

When you select **IDLE (Python GUI)** from the menu, you should see something like Figure 1 .

(Note to blind and visually impaired students: I don't know if you will find the Python Shell to be accessible. If not, don't worry. We will get to a command-line version that should be accessible a little later.)

Figure 1 . Python 3.4.1 Shell.


Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
Ln: 3 Col: 4

What is a GUI ?

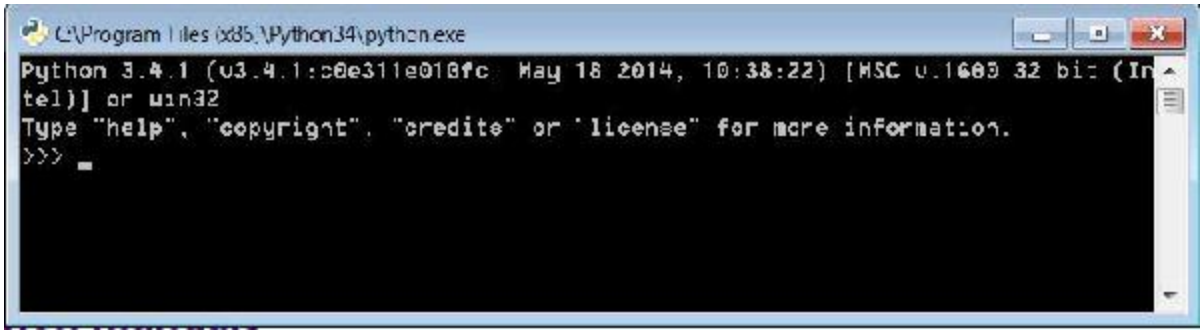
GUI is an acronym for *Graphical User Interface* . The IDLE window is a GUI. It can be used for interactive Python programming.

To make it go away when you are finished, simply click the button in the upper-right corner that is labeled with an X.

The Python (command line) selection

When I select the **Python (command line)** item in the menu mentioned above , I get essentially the same thing as the GUI, but in a "black screen" window commonly referred to as a command-line window as shown in Figure 2 .

Figure 2 . Python (command line) window.


C:\Program Files (x86)\Python34\python.exe
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

The command-line window

This window can also be used for interactive Python programming in much the same way that the Python Shell can be used. Each has some practical advantages and disadvantages.

To make this screen go away, click the X in the upper right-hand corner.

> Note: In addition to IDLE, many Python development environments are available on the web - some free and some not free. My personal favorite is the free Wing IDE . I have asked that it also be installed in the computer labs on the Northridge campus.

Your first (interactive) Python program

You can use IDLE, Wing *(if it is available)* , or the command line editor to write and execute your first Python program.

Hello World in Python

As has become the custom in programming circles, we will make our first Python program one that displays "Hello World" on the computer screen. We will write and execute it interactively.

The Python prompt >>>

The three right-angle brackets that you see in both of the interactive screens (>>>) make up the Python interactive prompt. When the cursor is blinking to the right of that prompt, you can enter a Python programming statement interactively.

Let's do it

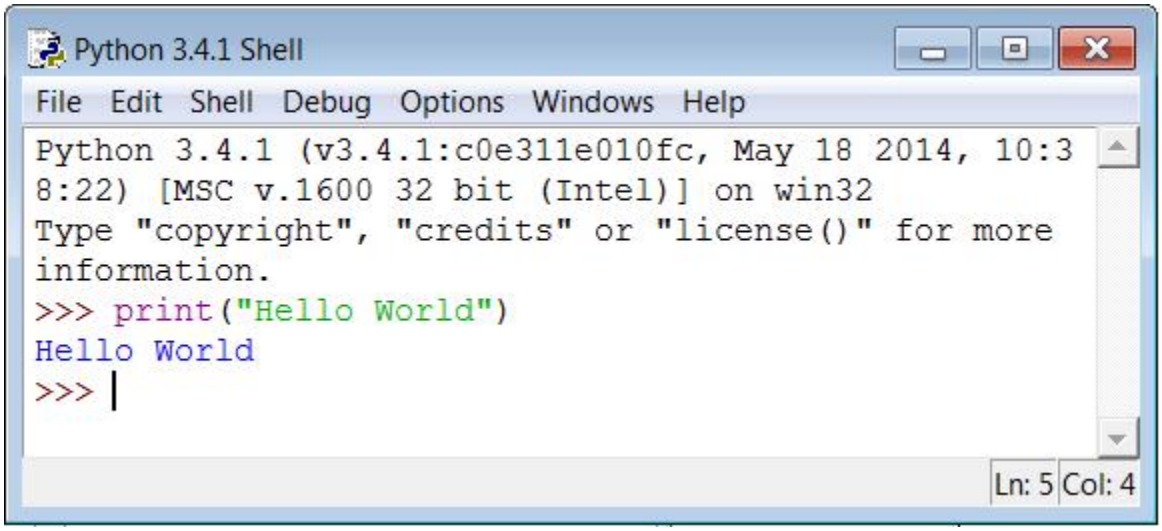
Type the following text to the right of the Python prompt and press the Enter key:

print("Hello World")

Note: The print function **:**
Prior to the release of version 3.0, print was a statement instead of a function and was executed without the parentheses.

If all goes well, your interactive Python screen should then look something like Figure 3 .

Figure 3 . Python Hello World.


Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:3
8:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
>>> print("Hello World")
Hello World
>>>
Ln: 5 Col: 4

Pay particular attention to the line that reads **Hello World** following your entry. That is the output from your program.

Congratulations

You have just written *(and executed)* your first *(interactive)* Python program, and possibly your first computer program as well. Not only that, you only had to type one line of code to write and execute your program.

Note that your entire program, the output from your program, and a new prompt are all shown in Figure 3 .

Python is ready for more

Python has provided a new prompt so that you can expand your program, or write another one.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1010-Getting Started
- File: Itse1359-1010.htm
- Published: 10/13/14
- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1015-Program Organization
This module provides a brief introduction to program organization using scripts, modules, and packages.

Table of contents

- Preface
- Discussion
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

You are probably reading this document online at cnx.org , otherwise known as **openstax cnx** . As of October 2014, two views of the document are available: an *openstax* view and a *legacy* view. In the legacy view, the document is commonly referred to as a *module* , which is contained in a *collection* of modules. I mention this here to distinguish the word *module* from the usage of the same word that you will find below .

This document provides a brief introduction to program organization using scripts, modules, and packages.

Discussion

Although you won't need this information initially, once your Python programs become larger than a few lines of code, you may **want to organize them into scripts** , **modules** , and **packages** . There is no shortage of online material on that topic. Here is a list of online resources that explain how to organize your programs:

- tutorialspoint -- Python Modules
- The Python Tutorial -- 6. Modules

- Python Course -- Modular Programming and Modules
- The original sthurlow.com python tutorial -- Modules
- learnpython.org -- Modules and Packages
- A Beginner's Python Tutorial/Importing Modules
- Non-Programmer's Tutorial for Python 3/Using Modules

Also, as you write more complicated programs, you will need to use modules that are contained in the standard Python distribution. Here is a link to an index of those modules in Python 3.4.2:

- Python Module Index

This information is being provided at this early stage in the course so that you will know where to find it when you need it.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1015-Program Organization
- File: Itse1359-1015.htm
- Published: 10/14/14
- Revised: 12/28/14

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1020r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1020-Numbers

Table of contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1020-Numbers* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? Python programming comments are ignored by the computer.

Go to answer 1

Question 2

True or False? Just like in C, C++, and Java, a Python comment begins with two slash characters (//) and continues to the end of the line.

Go to answer 2

Question 3

True or False? The only prompt used by the Python interactive system is the one consisting of three right-angle brackets (>>>).

Go to answer 3

Question 4

True or False? The output produced by the Python interactive system appears on a line without either of the prompts mentioned above.

Go to answer 4

Question 5

True or False? If you enter an expression at the prompt and press the Enter key, the result of evaluating the expression will be displayed on the next

line without a prompt.

Go to answer 5

Question 6

True or False? Computer programs are composed of expressions, which are made up of statements.

Go to answer 6

Question 7

In the following expression, 2+5, what is the common jargon for the plus sign, and what is the common jargon for the 2 and the 5?

Go to answer 7

Question 8

List the operators discussed in the module titled *Itse1359-1020-Numbers* , and describe the purpose of each.

Go to answer 8

Question 9

True or False? Integer division produces a decimal result with non-zero digits to the right of the decimal point.

Go to answer 9

Question 10

True or False? The modulus operator is used to produce the quotient in division.

Go to answer 10

Question 11

Describe the use of parentheses in expressions.

Go to answer 11

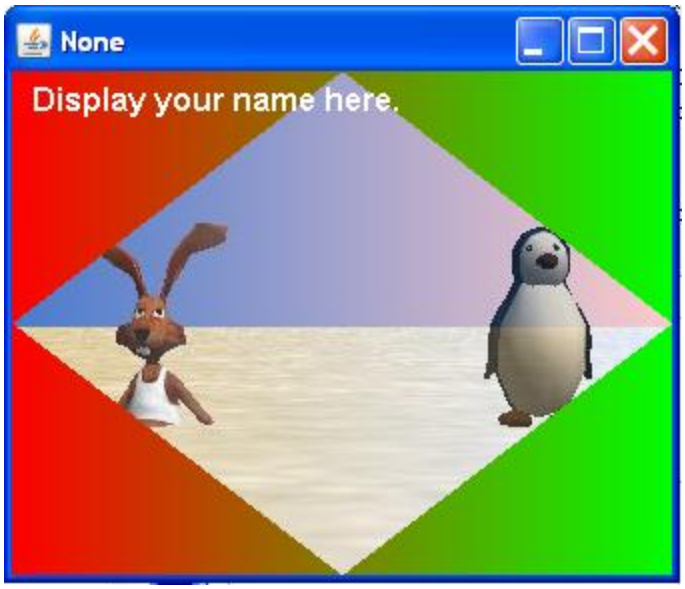
Question 12

Describe how Python evaluates an expression containing parentheses.

Go to answer 12

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 12

A Python expression is evaluated by first evaluating each of the sub-expressions in the parentheses, and then using those values to complete the

evaluation. If the expression contains nested parentheses, the evaluation is performed by evaluating the innermost parentheses first and working outwards from there.

Go back to Question 12

Answer 11

Parentheses can be used to group terms in an expression in order to provide control over the order in which the operations are performed.

Go back to Question 11

Answer 10

False, the modulus operator is used to produce the remainder.

Go back to Question 10

Answer 9

False. Integer division produces an integer or whole number result with no non-zero digits to the right of the decimal point even if a decimal point is displayed.

Go back to Question 9

Answer 8

The operators and their purposes are:

- The addition operator, +

- The subtraction operator, -
- The multiplication operator, *
- The division operators, / and //
- The modulus operator, %

Go back to Question 8

Answer 7

The plus sign is commonly called the operator. The 2 and the 5 are commonly called operands. More specifically, the 2 is the left operand and the 5 is the right operand.

Go back to Question 7

Answer 6

False. Just the reverse is true. Programs are made up of statements, which are composed of expressions.

Go back to Question 6

Answer 5

True, unless the expression can't be evaluated, in which case an error message will appear.

Go back to Question 5

Answer 4

True.

Go back to Question 4

Answer 3

False. The Python interactive system also uses a secondary prompt consisting of three periods (...).

Go back to Question 3

Answer 2

False. In Python, a comment starts with the hash character (#) and ends at the end of the line.

Go back to Question 2

Answer 1

True. Programming comments are used for program documentation and are intended for human consumption.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1020r-Review
- File: Itse1359-1020r.htm

- Published: 10/14/14
- Revised: 12/28/14

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1030r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1030-Variables and Identifiers.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1030-Variables and Identifiers* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? A variable is the same as a constant.

Go to answer 1

Question 2

True or False? Python is a *strongly typed* language.

Go to answer 2

Question 3

True or False? Python programmers must declare all variables.

Go to answer 3

Question 4

Explain the dangers of using a language that does not require variables to be declared.

Go to answer 4

Question 5

What is the best defense against spelling errors in variables names?

Go to answer 5

Question 6

True or False? Variable names can begin with numeric or digit characters.

Go to answer 6

Question 7

Write a simple program that illustrates case sensitivity in the names of variables.

Go to answer 7

Question 8

Explain the use of the assignment operator.

Go to answer 8

Question 9

Which type usually provides the greater range for storage of numeric values, integer or floating point?

Go to answer 9

Question 10

Should you just always use floating point instead of integer to be safe?

Go to answer 10

Question 11

Write a simple program that illustrates the approximation nature of floating point arithmetic.

Go to answer 11

Question 12

Explain the purpose of the automatic continuation variable whose name is simply the underscore character.

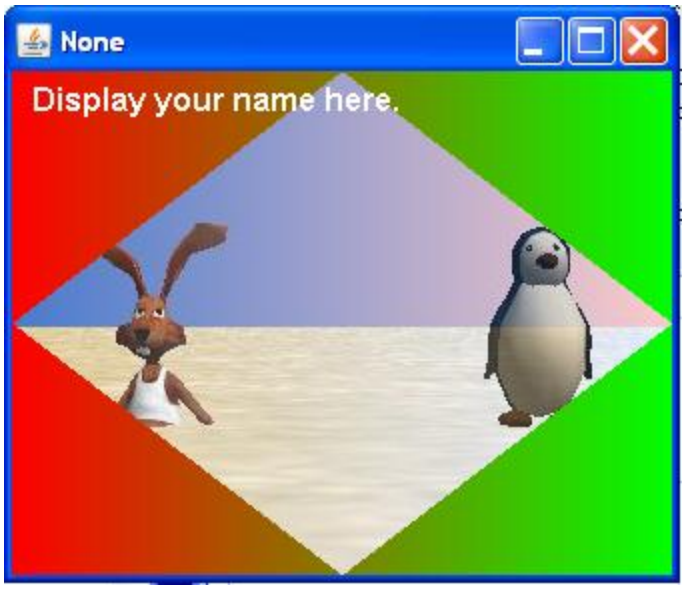
Go to answer 12

Figure index

- Figure 1 . Same letters, different case.
- Figure 2 . Same letters, same case.
- Figure 3 . Floating point errors.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 12

The primary purpose of the automatic variable named _ is to make it easier for you to string calculations together in interactive mode and to display the

intermediate results as you go.

Go back to Question 12

Answer 11

See the sample program in Figure 3 . We know that the true result of this expression is an unending string of nines, as in 9.999999999999999

Figure 3 . Floating point errors.

```
>>> 10/3 + 20/3
10.0
```

However, in this case, Python returned an answer of 10.0, indicating that the answer is accurate to three significant figures. This is not really the correct answer, but as a practical matter, it may be the *best* answer. I will leave that for you to decide.

Go back to Question 11

Answer 10

Probably not. Floating point arithmetic often suffers from speed penalties. In addition, integer arithmetic produces exact results while floating point

arithmetic usually produces approximate results *(although the approximations may be very close to being exact in many cases)* .

Go back to Question 10

Answer 9

Floating point usually provides the greater range for storage of numeric values.

Go back to Question 9

Answer 8

The assignment operator causes the value of its right operand to be stored in the memory location identified by its left operand.

Go back to Question 8

Answer 7

An example of such a program is shown in Figure 1 . Note that the names of the two variables have the same letters, but different case. The fact that the two variables are different variables is illustrated by the fact that each is assigned a different value. The sum of the two variables demonstrates that the two variables contain different, and correct, values.

Figure 1 . Same letters, different case.

Figure 1 . Same letters, different case.

```
>>> aX=10 # this is one variable
>>> Ax=20 # this is a different variable
>>> aX+Ax
30
>>>
```

Contrast the above result with the program in Figure 2 where a variable whose name contains the same letters and the same case is used twice.

Figure 2 . Same letters, same case.

```
>>> ax=10
>>> ax=20
>>> ax+ax
40
>>>
```

All the program in Figure 2 accomplishes is the assignment of two different values to the same variable. The second assignment overwrites the value previously assigned to the variable. The program then adds the variable to itself using its current value of 20 producing a result of 40 *(instead of 30 as in the previous example)* .

Go back to Question 7

Answer 6

False. Variable names must begin with a letter or underscore character.

Go back to Question 6

Answer 5

Use meaningful variable names for which the spelling is obvious, such as **theOverheadRate** .

Go back to Question 5

Answer 4

Several different kinds of problems can result from making typing errors that result in misspelling the names of variables. These errors usually result in programs that produce incorrect results without warning.

Go back to Question 4

Answer 3

False. Variable declarations are not required in Python. All that is required to cause a variable to come into existence is to invent a new name for a variable and assign a value to it.

Go back to Question 3

Answer 2

False. Python is not a strongly typed language. From a pure technical viewpoint, the Python programmer rarely needs to be concerned about type.

Go back to Question 2

Answer 1

False. The value of a variable is intended to change during the execution of the program. The value of a constant *(which I haven't discussed so far in this collection)* is not intended to change.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1030r-Review
- File: Itse1359-1030r.htm
- Published: 10/14/14
- Revised: 20/20/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1050-Introduction to Scripts
This module provides an introduction to the use of scripts in Python.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
    - Important characteristics of scripts
        - Scripts are reusable
        - Scripts are editable
    - You will need a text editor
    - Combining scripts with interactive mode
- Getting started
    - The location of python.exe
        - Setting the path permanently
        - Setting the path temporarily
            - A Windows batch file
            - Setting the path with a batch file
    - Create a script file
        - Run your script file
        - The correct output
    - Alternative approaches

        - Skulpt
        - Online code visualizer
        - How To Think Like a Computer Scientist

    - Visualize your script file with the Online Python Tutor

- What's next?

    - Practice, practice
    - Cut-and-paste programming

- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It provides an introduction to the use of scripts, and of necessity will depart from the interactive mode used in previous modules.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. However, there are a couple of bitmap images that may not be accessible.)

Figures

- Figure 1 . Screen output from running the script named 1359-1050-01.py.
- Figure 2 . Online Python Tutor visualizer screen output.
- Figure 3 . Visualizing a program from a future module.

Listings

- Listing 1 . Setting the path temporarily with a batch file.
- Listing 2 . Contents of the file named 1359-1050-01.py.

Introduction

This module provides an introduction to the use of scripts, and of necessity will depart from the interactive mode used in previous modules.

It will be necessary for me to get into a small amount of *system stuff* that really has nothing in particular to do with Python programming. Rather, it will involve getting your computer set up for using scripts with Python.

Up to this point in this collection, I have concentrated on the interactive programming capability of Python. This is a very useful capability that allows you to type in a program and to have it executed immediately in an interactive mode. But, interactive can be burdensome. By now you may have realized that you sometimes find yourself typing the same thing over and over. That is where scripts are useful.

Important characteristics of scripts

Scripts are reusable

Basically, a script is a text file containing the statements that comprise a Python program. Once you have created the script, you can execute it over and over without having to retype it each time.

Scripts are editable

Perhaps, more importantly, you can make different versions of the script by modifying the statements from one file to the next using a text editor. Then you can execute each of the individual versions. In this way, it is easy to create different programs with a minimum amount of typing.

You will need a text editor

Just about any text editor will suffice for creating Python script files and you may already have a favorite. Whichever editor you choose, make certain that it produces plain ASCII text in its output *(no bold, no underline, no italics, etc.)* .

In addition to functioning as an interactive interface, the IDLE GUI interface also provides a text editor for creating scripts. Once you have the GUI interface open, select **New File** from the **File** menu. After that, the behavior is similar to many other text editors.

Combining scripts with interactive mode

It is also possible to combine script files with interactive mode to incorporate pre-written scripts into interactive programs.

Getting started

In order to use script files, you must prepare you computer to use them or you must pursue one of the alternative approaches described later. Preparing your computer for use with Python doesn't amount to much in the way of effort, but it is critical. All of the following instructions are based on the Windows operating system. If you are using some other operating system, you will need to translate the instructions for use with that operating system.

The location of python.exe

When you first installed your Python software, a directory should have been created somewhere on your hard drive containing a file named **python.exe** . You will need to locate that file. On my Windows machine, the file is located in the following folder:

C:\Program Files (x86)\Python34

Setting the path permanently

In case you are unfamiliar with the term, the *"path"* is a text string, otherwise known as an *environment variable* , that tells the operating system where to search for executable files such as the file named **python.exe** .

You will need to cause the directory containing the file named **python.exe** to be listed in your *system environment variable* named *path* , either permanently or temporarily.

If you already know how to set the path, and you want to make a permanent change to the path, go ahead and do it. If you don't already know how, you may need to get some help.

I'm not going to try to tell you how to make a permanent change to the path. If you don't do it correctly, you may cause problems that are difficult to recover from. I don't want to be responsible for that. However, I will tell you how to make a temporary change to the path in the next section.

Setting the path temporarily

A Windows batch file

A Windows batch file is an executable text file with an extension of .bat. In its simplest form, it contains one or more commands that you would otherwise enter, one at a time, at the command-line prompt. When you execute the batch file, it attempts to cause those commands to be executed sequentially and automatically.

I am a big believer in the use of batch files for a variety of reasons, not the least of which is the ability to type a set of commands once and to execute them many times without having to retype them.

Batch files also make it possible to execute commands on the command line in the "current directory" without the requirement to go through the steps that are normally required to open a command-line window and navigate to the current directory.

Setting the path with a batch file

Listing 1 shows the contents of a batch file named **1359-1050-01.bat** containing four commands that :

1. Reduce the clutter on the output screen *(echo off)* .
2. Temporarily set the path to the location of the file named **python.exe** on my computer.
3. Attempt to execute the Python script file named 1359-1050-01.py, which will be discussed later in this module.
4. Pause onscreen and wait to be dismissed with the prompt *"Press any key to continue . . ."*

Listing 1 . Setting the path temporarily with a batch file.

Listing 1 . Setting the path temporarily with a batch file.

```
echo off

path=%path%;"C:\Program Files (x86)\Python34"

python 1359-1050-01.py

pause
```

The second command in Listing 1 is the important command for this section of this module. It will temporarily set the path to the indicated folder. If you elect to use this approach, you will need to replace the text inside the double quotes with the path to the folder containing the file named **python.exe** on your computer.

This approach appends the new path onto the environment variable named *path* on a temporary basis. It goes away as soon as you respond to the **pause** statement at the end or otherwise close the command-line window that appears when you execute the batch file.

Create a script file

Once you have the path variable properly set, or you know how to set it temporarily in a batch file, use the text editor of your choice and create a file named **1359-1050-01.py** that contains the Python programming statements shown in Listing 2 . *(You can give the file any name that you choose provided that the extension is .py and provided that you properly reflect the name in the third line of the batch file shown in Listing 1 .)*

Listing 2 . Contents of the file named 1359-1050-01.py.

```
a=2
b=3
a=2*a
b=3*b
c=a+b
print(c)
```

You should recognize all of the statements in Listing 2 . Note, however, that in a script, they are not preceded by an interactive prompt (>>>).

Store this file in any folder on your hard drive. You may want to create a new directory for the sole purpose of storing Python script files.

Then create the corresponding batch file with the contents shown in Listing 1 and save it in the same folder as the script file. *(If you have permanently set your path to support Python, you can omit the path command in Listing 1 .)*

Run your script file

Once you have accomplished the above, double-click the batch file to execute it. The batch file will temporarily set the path and attempt to run your script. If all goes well, a command-line window will appear on your screen with contents similar to that shown in Figure 1 . *(Note that I deleted some of the text on the first line for brevity. Your first line will be different anyway.)*

Figure 1 . Screen output from running the script named 1359-1050-01.py.

```
M:\Baldwin\AA-School\...>echo off
13
Press any key to continue . . .
```

The correct output

If you go back to the script file in Listing 2 and do the arithmetic, you will see that the output value of 13 produced by the program is correct.

If you got an output value of 13, -- congratulations -- you have just written and executed your first Python script. If not, try to figure out why.

Alternative approaches

There are at least three *(and probably more)* alternative online approaches that allow you to write and run Python code without a requirement to install Python on your computer:

- Skulpt
- Online code visualizer
- How To Think Like a Computer Scientist - Interactive

All three of these approaches run in a browser and can even be used with a tablet or a smart phone.

Skulpt

Skulpt is an entirely in-browser implementation of Python. No local installation, plugins, or server-side support is required.

Just go to www.skulpt.org in your computer, tablet, or smart phone, scroll down to the **Demo** pane, type in your Python code and click the **Run** button below the **Demo** pane. The program output will appear in the **Output** pane below the **Run** button.

You can learn more about using Skulpt, including how to embed Skulpt in your HTML pages in the module titled Itse1359-2410-Getting Started with Skulpt in this book .

Online code visualizer

See the section titled Visualize your script file with the Online Python Tutor later in this module for a discussion of an online code visualizer. This tool also runs in a browser and can be used with a computer, a tablet, or a smart phone.

How To Think Like a Computer Scientist

How To Think Like a Computer Scientist - Interactive is an online interactive textbook that can be freely accessed in your browser on a computer, a tablet, or a smart phone. It uses the two tools described above plus video, self-check quizzes, and other features to teach Python programming. There are numerous opportunities within the book for you to write and run Python code completely within your browser.

(Note that as of February, 2015, the online video in the book seems to have an audio compatibility problem with my Android tablet. I have discussed this with the author and I expect the problem to be corrected soon.)

Visualize your script file with the Online Python Tutor

When learning to program, it is often useful to be able to follow the execution of a program step-by-step. A free online tool called the Online Python Tutor makes this possible. *(You have seen some output from this tool in earlier modules. I will provide a more detailed explanation of how to use the tool in the module titled Itse1359-1065-Visualizing Python in this book .)*

The following quotation was copied from pythontutor.com .

> *" Online Python Tutor is a free educational tool created by Philip Guo that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write **Python** , **Java** , and **JavaScript** programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs."*

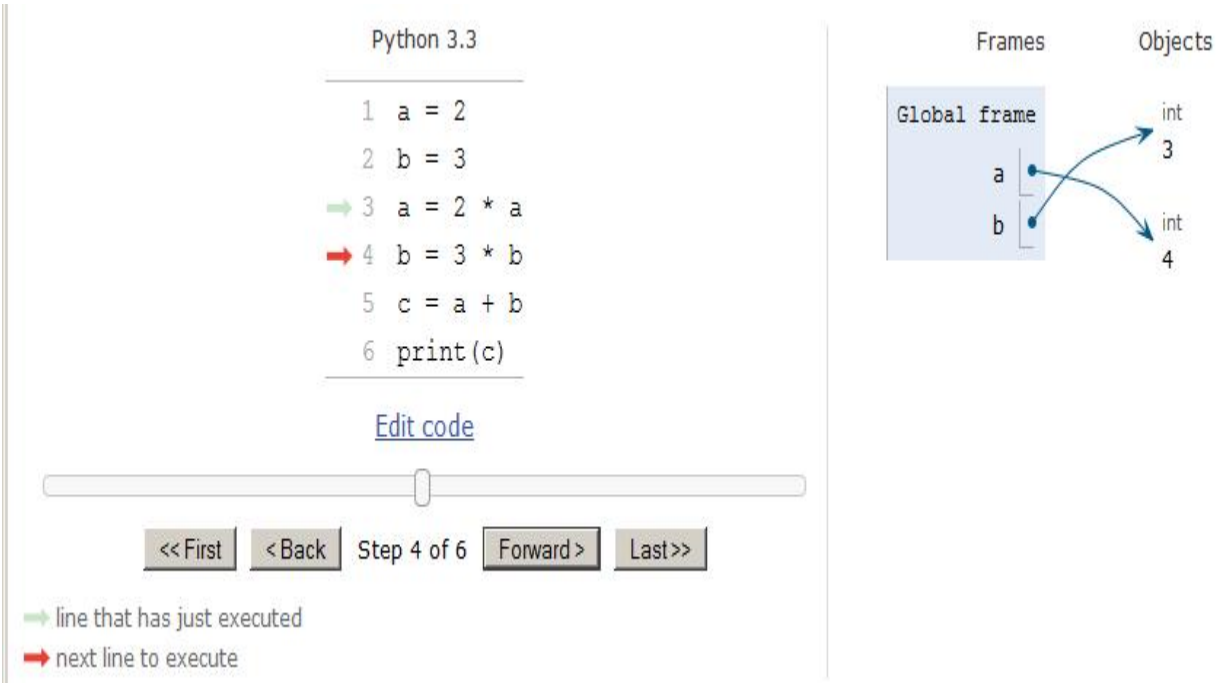
Beginning with this module and continuing with future modules, I will recommend that you

- Open the Online Python Tutor .
- Copy the example programs, such as the code shown in Listing 2 , into the online text editor .
- Select Python 3.3 in the pull-down list.
- Click the button labeled **Visualize Execution** .

The screen will then change to include what you see in Figure 2 . Use the **Forward** and **Back** buttons to step forward and backward through your code, one instruction at a time, while observing the information that appears on the screen to the right of the code window.

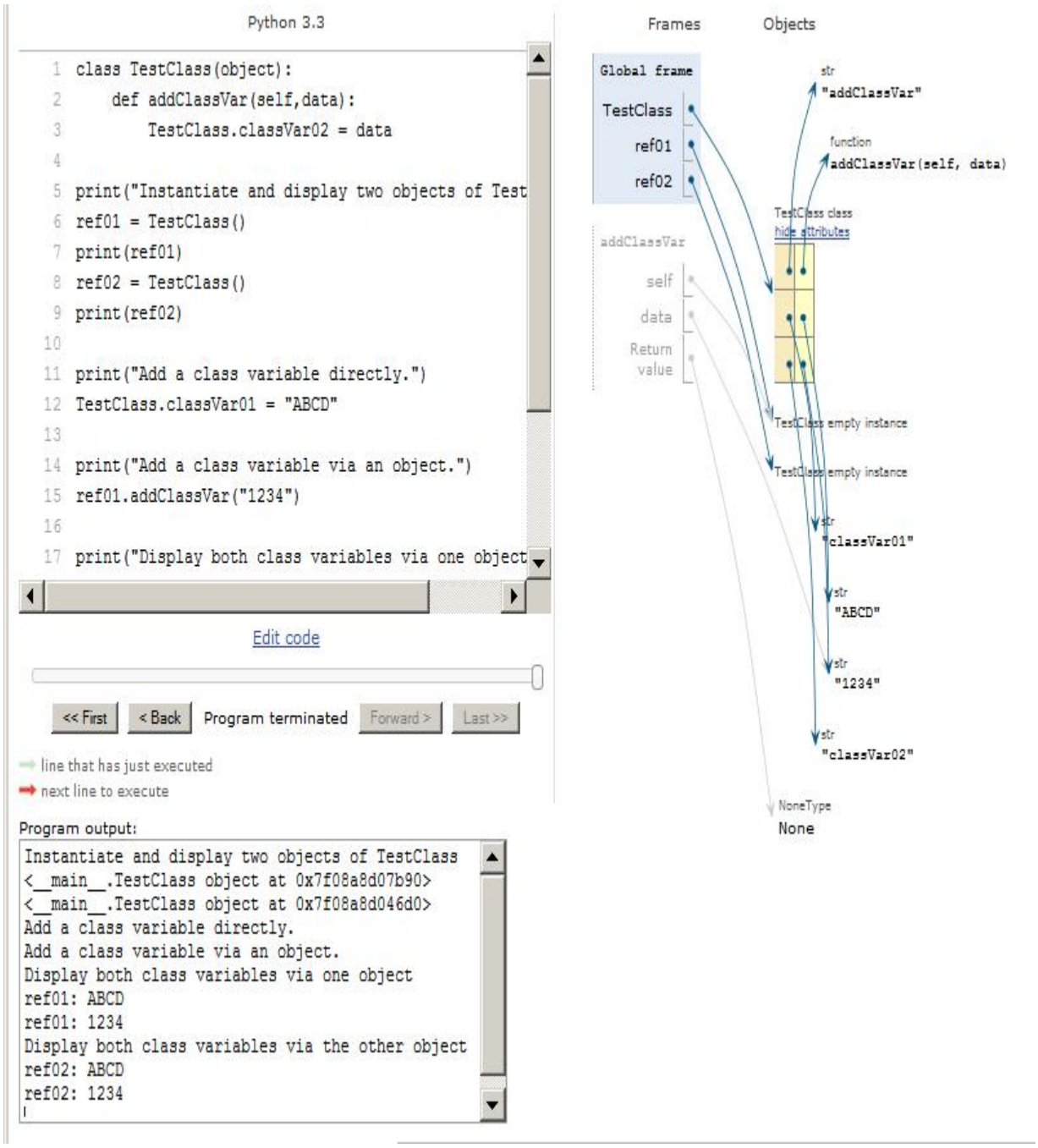
For example, Figure 2 shows the Online Python Tutor visualizer screen after the first three statements from Listing 2 have been executed. The information on the right shows the values stored in the variables named **a** and **b** at that point in the execution of the program.

Figure 2 . Online Python Tutor visualizer screen output.



As you might expect, more complicated code produces more complicated output from the visualizer. For example, Figure 3 shows the output from the Online Python Tutor when visualizing a program from a future module titled Itse1359-1440-Class Variables in this book .

Figure 3 . Visualizing a program from a future module.



What's next ?

Obviously, there is a lot more that you will need to learn before you can write that "killer app" that takes the world by storm, but at this point, you have the tools to experiment with some simple scripts.

Practice, practice

I recommend that you look back into the earlier modules and convert some of the interactive programs listed there into scripts, execute them, and confirm that the scripts behave as expected.

Cut-and-paste programming

In future modules, I will switch back and forth between scripts and interactive mode, depending on which seems to be the most appropriate at the time.

I am a strong advocate of cut-and-paste programming. Cut-and-paste programming works well with scripts, and not so well with interactive mode.

Therefore, any time there is very much typing involved, you can usually expect to see me using scripts instead of interactive mode.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1050-Introduction to Scripts
- File: Itse1359-1050.htm
- Published: 10/14/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1060-Syntax
This module will help you to understand the physical construction of a Python program with particular emphasis on indentation.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- Program construction
    - A physical line
    - A logical line
        - Statements and logical lines
        - Comments
    - Explicit line joining
        - The backslash character
        - For illustration only
        - No spaces or comments allowed
        - Otherwise illegal
    - Implicit line joining
    - Blank lines
- Indentation
    - Indentation is used to determine grouping of statements
        - A blessing and a curse
        - No safety nets
    - An example of correct indentation
        - Compare A with B and take appropriate action
        - A compound statement
        - Group behavior
        - Another sample script
        - Members of the group
        - An opinion
        - Indentation details

        - Leading whitespace
        - Tabs
    - The bottom line on indentation
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It explains some of the mechanics of putting together a Python program.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Two physical lines.
- Figure 2 . Output from the script in Listing 4.

Listings

- Listing 1 . Explicit line joining.
- Listing 2 . Implicit line joining.
- Listing 3 . Blank lines and whitespace.
- Listing 4 . An example of correct indentation.
- Listing 5 . Another sample script.

Introduction

You have learned how to write some simple programs and how to execute them interactively.

You have also learned how to capture simple programs in script files and how to execute those script files.

Now it is time to learn a little more about the mechanics of putting together a Python program.

Program construction

Conceptually, programs are constructed from statements, and statements are composed of expressions. In practice, you need to know how to construct statements from a physical viewpoint.

A Python program is divided into a number of *logical lines* . A logical line is constructed from one or more *physical lines.*

A physical line

A physical line ends with the character or characters used by your platform for terminating lines.

On Unix, this is the *linefeed* character. On Windows, it is the *carriage return* character followed by the *linefeed* character. On Macintosh, it is the *carriage return* character.

We refer to this in programming jargon as the *newline* character *(even though it is actually two characters on Windows systems)* .

Basically, a physical line is what you get when you type some characters into your text editor and press the *Enter* key, the *Return* key, or whatever you call that key on your keyboard. *(Actually you don't even need to type characters before pressing the Enter key, in which case you get a blank line.)*

This is the key that causes the cursor to go down to the next line and return to the left side of the editing window.

For example, when creating the text shown in Figure 1 using a text editor on my Windows computer, I pressed the Enter key immediately following the --> for each line. This produced a *newline* in each case.

Figure 1 . Two physical lines.

```
I will end the line here -->
and end the next line here -->
```

A logical line

A *logical line* may be the same as a *physical line* , or line joining rules *(described later)* can be used to construct a logical line from two or more physical lines.

Statements and logical lines

A statement cannot cross logical line boundaries except where the syntax allows for the *newline* character, such as in *compound statements* . I will show you an example of a compound statement later in this module.

Comments

We learned about comments in an earlier module. To summarize, a comment starts with a hash character (#) that is not part of a string literal *(we also learned about string literals in an earlier module)* . The comment ends at the end of the physical line.

Explicit line joining

You can join two or more *physical lines* to produce a *logical line* , using the backslash character as shown in Listing 1 .

Listing 1 . Explicit line joining.

```
a = 3
b = 4
# construct c=a+b
c \
= \
a \
+ \
b
print(c)

# the output is 7
```

The backslash character

When a *physical line* ends in a backslash that is not part of a string literal or comment, that line is joined with the following *physical line* forming a single *logical line* .

The backslash and the following end-of-line character(s) are ignored by the interpreter and do not become part of the logical line.

In Listing 1 , the expression **c=a+b** is created by joining five consecutive *physical lines* to create one *logical line* .

For illustration only

Obviously, this is not how you would want to write a long script, but it is not unusual to break long expressions into two or more physical lines to make them fit onto a prescribed page width.

No spaces or comments allowed

You must make certain that no space characters follow the backslash.

You may not place a comment following the backslash, and a backslash does not continue a comment.

Otherwise illegal

A backslash is illegal elsewhere on a line except inside a string literal.

Implicit line joining

Expressions in parentheses, square brackets, or curly brackets can be split over more than one physical line without using backslashes as shown in Listing 2 .

Listing 2 . Implicit line joining.

Listing 2 . Implicit line joining.

```
a = 3
b = 4
c = (a # continue on next line
     + b)
print(c)

# the output is 7
```

You can also place a comment on a line that is being continued implicitly as shown in the third line of Listing 2 .

Also, you can indent the continuation line however you want. This is very useful for making the code more readable.

Blank lines

Lines containing only spaces, tabs, form feeds, and comments are ignored in scripts, but the behavior may be different in interactive mode, depending on the implementation.

An example of some spaces and blank lines is shown in Listing 3 .

Listing 3 . Blank lines and whitespace.

Listing 3 . Blank lines and whitespace.

```
a = 3
  # spaces and comment
b = 4
    # tab and comment
c = (a # continue on next line
     + b)
# a blank line follows

print(c)

# the output is 7
```

Indentation

In every other programming environment that I have worked with in the past, indentation is used strictly for cosmetic or readability purposes.

However, indentation is not used strictly for cosmetic or readability purposes in Python. It is such an important topic in Python that I have separated it into a major section in this module.

Indentation is used to determine grouping of statements

Unlike most other programming environments, physical indentation is used in Python to determine the grouping of statements.

A blessing and a curse

This can be both a blessing and a curse. The blessing is that it forces you to use proper indentation, which usually leads to more readable code.

The curse is that if you don't use proper indentation, your script probably won't behave properly.

No safety nets

There are few if any safety nets in Python *(such as the matching curly brackets used in C, C++, and Java)* to protect you from indentation errors.

To make matters worse, such errors often turn up as logical errors *(meaning that the script simply doesn't work correctly)* rather than interpreter errors *(meaning that the interpreter will tell you about the error)* .

An example of correct indentation

Although I haven't introduced you to the **if** statement yet, you probably have a fairly good idea what it is used for. I am going to use it to illustrate the proper indentation of a *group of statements* with the script in Listing 4 .

Note: Description of an "if" statement:
An **if** statement means that if some expression evaluates to **true** , the program will do something specific. Otherwise, the script won't do that specific thing.

Don't be too concerned if the logic of the script in Listing 4 escapes you at this point in time. I will explain the use of the **if** statement in detail in a future module. The important thing here is to understand the grouping of statements.

Listing 4 . An example of correct indentation.

Listing 4 . An example of correct indentation.

```
A = 3
B = 4

if B > A:
  print(A) # begin group
  print(B)
  print(A + B) # end group
A = 6 # not part of above group
print(A)

#============================================================
#The output, which is not part of the script, is shown below.
3
4
7
6
```

Compare A with B and take appropriate action

The script in Listing 4 compares the value of the variable **A** with the value of the variable **B** to determine if the value of **B** is greater than the value of **A** *(if B > A:)* .

If the value of **B** is greater than the value of **A** *(which it is in this case)* , the three indented print statements are executed. Otherwise, that group of three statements is bypassed.

A compound statement

The three statements shown with indentation are either all executed, or they are all bypassed. Hence, they behave as a group.

A group of statements like this is sometimes referred to as a *compound statement (a statement made up of two or more individual statements)* .

Group behavior

When the three indented statements are executed as a group, the values 3, 4, and 7 are printed on consecutive output lines by the three print statements in the group of statements, as shown in Figure 2 .

Figure 2 . Output from the script in Listing 4.

```
C:\jnk\6>echo off
3
4
7
6
Press any key to continue . . .
```

Then the value 6 is printed by the last statement in the script, which is not part of the group. *(Note that the last statement is not indented.)*

Another sample script

Now, let's make a change to the script. The script shown in Listing 5 is identical to the one in Listing 4 except that I switched the values of **A** and **B** to cause the group of indented statements to be bypassed *(**B** is no longer greater than **A**)*.

Listing 5 . Another sample script.

```
A = 4
B = 3

if B > A:
  print(A) # begin group
  print(B)
  print(A + B) # end group
A = 6 # not part of above group
print(A)

#============================================================
#The output, which is not part of the script, is shown below.
6
```

In this case, only one value (6) is printed because the three print statements in the group were bypassed as a group.

Members of the group

The important point here is that the three indented statements in Listing 4 and Listing 5 constitute a group because of their *common indentation level* .

An opinion

I personally don't like the idea of using indentation to create grouping. Although it sounds nice in theory, it can be very labor intensive in practice. Once you have written a script, one simple change can often require you to go back and modify the indentation level of almost every statement in the script.

I guess the good news is that this will encourage you to write your script as a series of short, concise independent modules rather than as a single long rambling script.

I am also concerned about the accessibility or lack thereof that grouping based on indentation level provides for blind and visually impaired students.

Indentation details

Leading whitespace

Leading whitespace *(spaces and tabs)* at the beginning of a *logical line* is used to compute the indentation level of the line. This, in turn, is used to determine the grouping of statements.

Tabs

My advice is to avoid the use of tabs altogether. Use spaces instead, and use the same number of spaces for each statement in the group.

However, if you must use tabs, you should go to the Python Language Reference -- 2.1.8. Indentation and make certain that you understand how Python deals with tabs.

The bottom line on indentation

Use spaces to cause the indentation level of all statements in a group of statements to be indented to the same level. Statements in a group are either all executed, or all bypassed by the

program logic.

I will provide a visualization of the behavior of indentation in the module titled Itse1359-1065-Visualizing Python in this book .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1060-Syntax
- File: Itse1359-1060.htm
- Published: 10/14/14
- Revised: 03/26/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1065-Visualizing Python
This module explains how to use a free educational code visualizer tool created by Philip Guo.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- General background information
- Recommendation
- Discussion and sample code
    - Open the online text editor
    - Three drop-down boxes
    - Beginning the visualization process
    - The red and green arrows
    - Output following three presses of the Forward button
    - Same situation, different code
    - Stepping through the code
    - The student torture program
    - Sharing the code visualizer with a tutor
- Run the programs
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on the use of the Online Python Tutor , which includes a free educational *code visualizer tool* created by Philip Guo .

This code visualizer tool helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this code visualizer tool, a teacher or student can write Python, Java, and JavaScript programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- Figure 1. The online text editor.
- Figure 2. Beginning the visualization process.
- Figure 3. Output following three presses of the Forward button.
- Figure 4. Same situation, different code.
- Figure 5. The student torture program.

Listings

- Listing 1. A sample program with proper indentation.
- Listing 2. Another sample program with proper indentation.

General background information

I touched on the use of the code visualizer in the earlier module titled Itse1359-1050-Introduction to Scripts . I also touched on the code visualizer in the module titled Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer in this book . The purpose of this module is to provide additional information related to the code visualizer.

When learning to program, it is often useful to be able to follow the execution of a program step-by-step. A free online tool called the Online Python Tutor makes this possible. The following quotation was copied from pythontutor.com .

> *" Online Python Tutor is a free educational tool created by Philip Guo that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write **Python** , **Java** , and **JavaScript** programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs."*

Recommendation

I recommend that from this point forward in this course, you copy and paste every sample program into the code visualizer and observe the behavior of that code as you step through the program, one instruction at a time. I believe that doing that will help you to better understand Python programming.

The procedure for doing that is as follows:

- Open the code visualizer .
- Copy the example program code into the online text editor .
- Select Python 3.3 in the pull-down list.
- Click the button labeled **Visualize Execution** .
- Use the **Forward** and **Back** buttons to step forward and backward through your code, one instruction at a time, while observing the positions of the red and green arrows and observing the information that appears on the screen to the right of the code window.

Discussion and sample code

The earlier module titled Itse1359-1060-Syntax in this book presented and explained the two Python programs shown in Listing 1 and Listing 2 . I will illustrate the use of the code visualizer with this pair of sample programs.

Listing 1. A sample program with proper indentation.

```
A = 3
B = 4

if B > A:
  print(A) # begin group
  print(B)
  print(A + B) # end group
A = 6 # not part of above group
print(A)
```

Listing 2. Another sample program with proper indentation.

Listing 2. Another sample program with proper indentation.

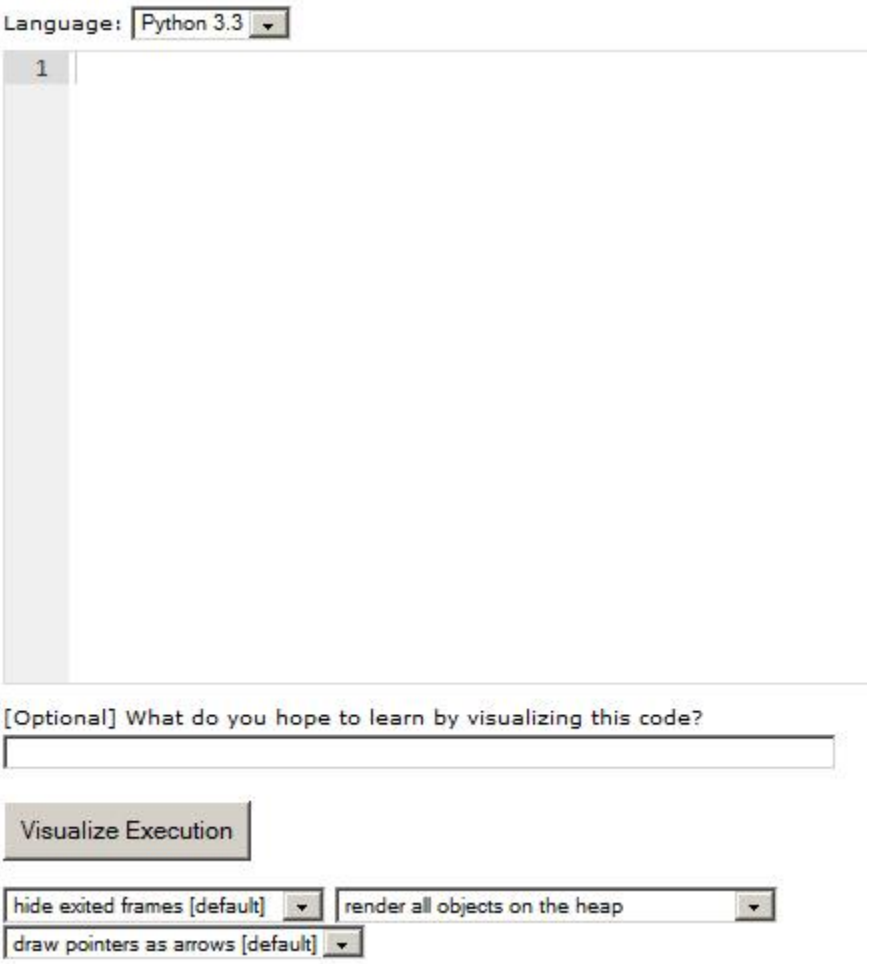
```
A = 4
B = 3

if B > A:
  print(A) # begin group
  print(B)
  print(A + B) # end group
A = 6 # not part of above group
print(A)
```

Open the online text editor

When you first open the online text editor , a portion of the screen will look like Figure 1 .

Figure 1. The online text editor.


Language: Python 3.3
1
[Optional] What do you hope to learn by visualizing this code?
Visualize Execution
hide exited frames [default]
render all objects on the heap
draw pointers as arrows [default]

The blank portion at the top of Figure 1 is the text editor portion of the screen.

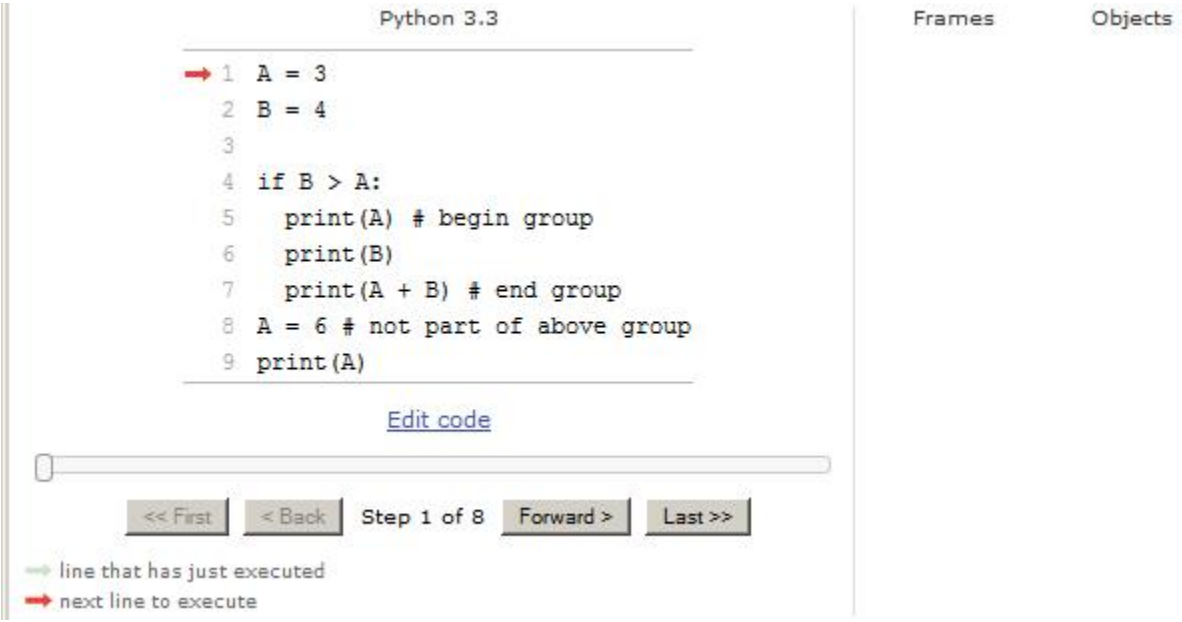
Three drop-down boxes

There are three drop-down boxes at the bottom of Figure 1 that allow you to select different options. The different options in these boxes control the manner in which the output information is presented as you step through the program. In some cases, you might want to step through the same program more than once with different sets of options selected.

Beginning the visualization process

Figure 2 shows the result of pasting the code from Listing 1 into the text editor and pressing the button in Figure 1 labeled **Visualize Execution** .

Figure 2. Beginning the visualization process.



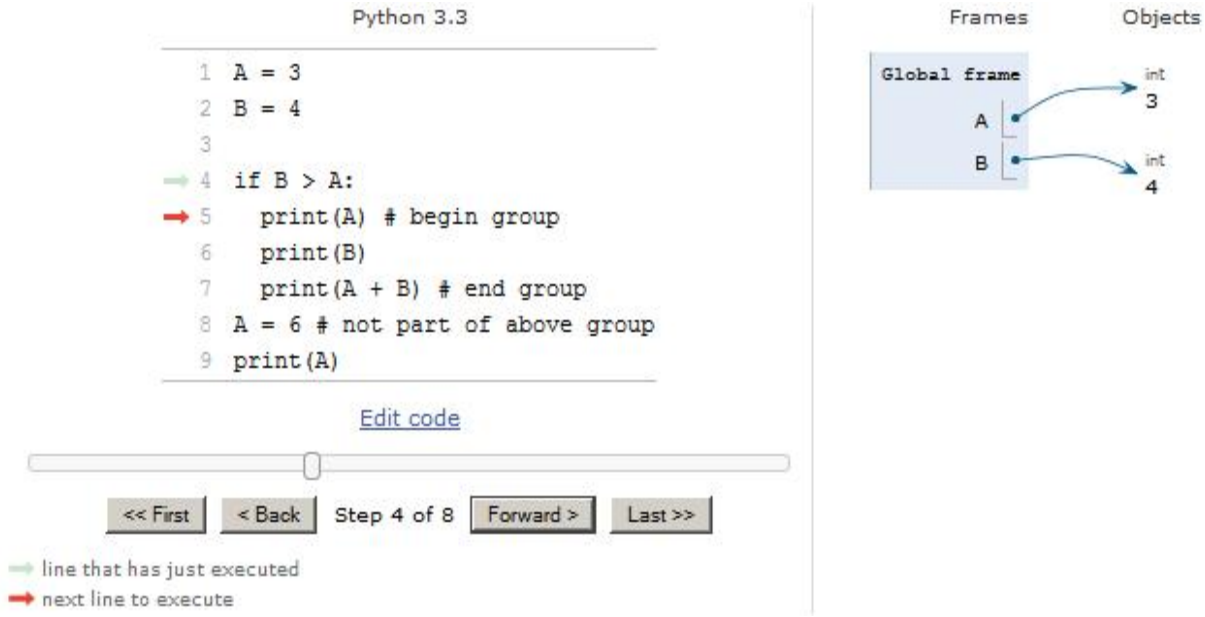
The red and green arrows

As indicated by the key in the bottom left corner of Figure 2 , two arrows will appear next to the code as you step through the code. A red arrow shows the instruction that will be executed the next time you press the **Forward** button. *(Pressing the **Back** button moves the red arrow back one step.)* The green arrow shows the instruction that was most recently executed. *(The green arrow isn't visible in Figure 2 because it is underneath the red arrow at the beginning.)*

Output following three presses of the Forward button

Figure 3 shows the result of pressing the **Forward** button three times.

Figure 3. Output following three presses of the Forward button.



We see from the green arrow on the left that the relational test in the **if** statement was the most recently executed instruction.

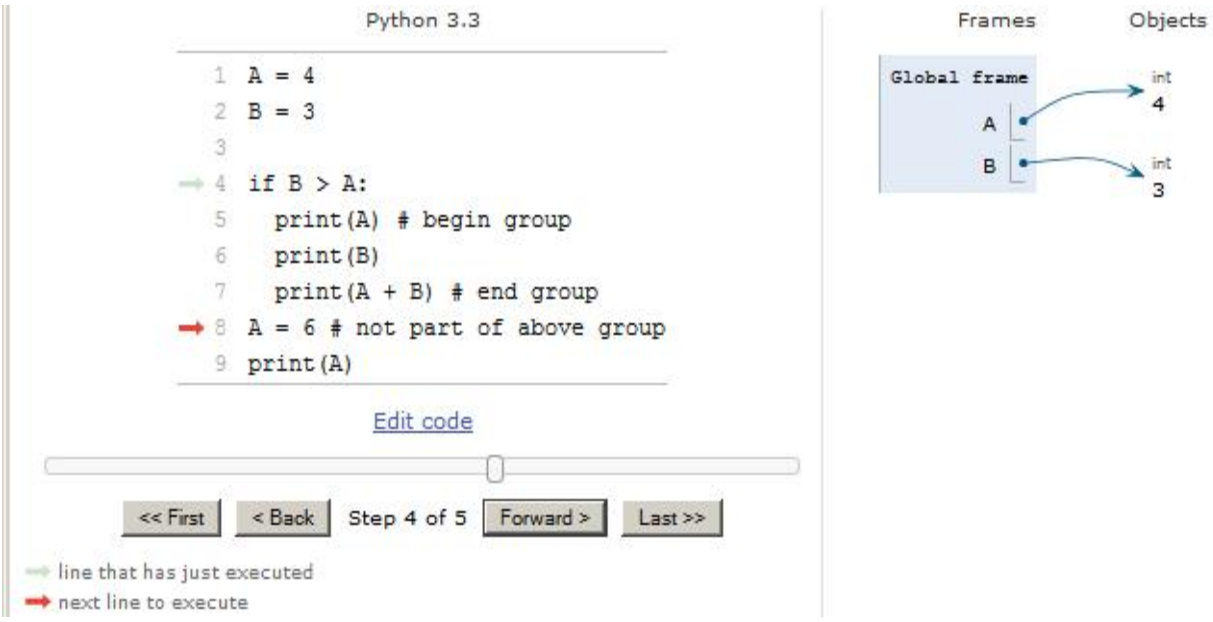
We see on the right that the variables named **A** and **B** contain references *(pointers)* to objects of type **int** containing values of **3** and **4** respectively.

Because **B** is greater than **A** , the relational test in the **if** statement *(pointed to by the green arrow)* returned true when that statement was executed. This means that the three statements in the indented block following the **if** statement will be executed. The red arrow points to the first statement in the indented block meaning that it will be executed the next time the user presses the **Forward** button.

Same situation, different code

Figure 4 shows the same situation using the code from Listing 2. instead of the code from Listing 1 .

Figure 4. Same situation, different code.



Note that the values stored in the variables named **A** and **B** were reversed relative to the code in Figure 3 . As a result, the **if** statement pointed to by the green arrow returned false instead of true when that statement was executed. Because of that, execution of the indented code block following the **if** statement will be skipped entirely and the red arrow now points to the statement immediately following that code block. That is the next statement that will be executed.

Stepping through the code

As you use the **Forward** and **Back** buttons to step through the code, one instruction at a time, you can observe the red and green arrows to gain a better understanding of the flow of control through the program. *(You will learn more about flow of control in future modules and you will probably find that the code visualizer is particularly useful when analyzing that code.)*

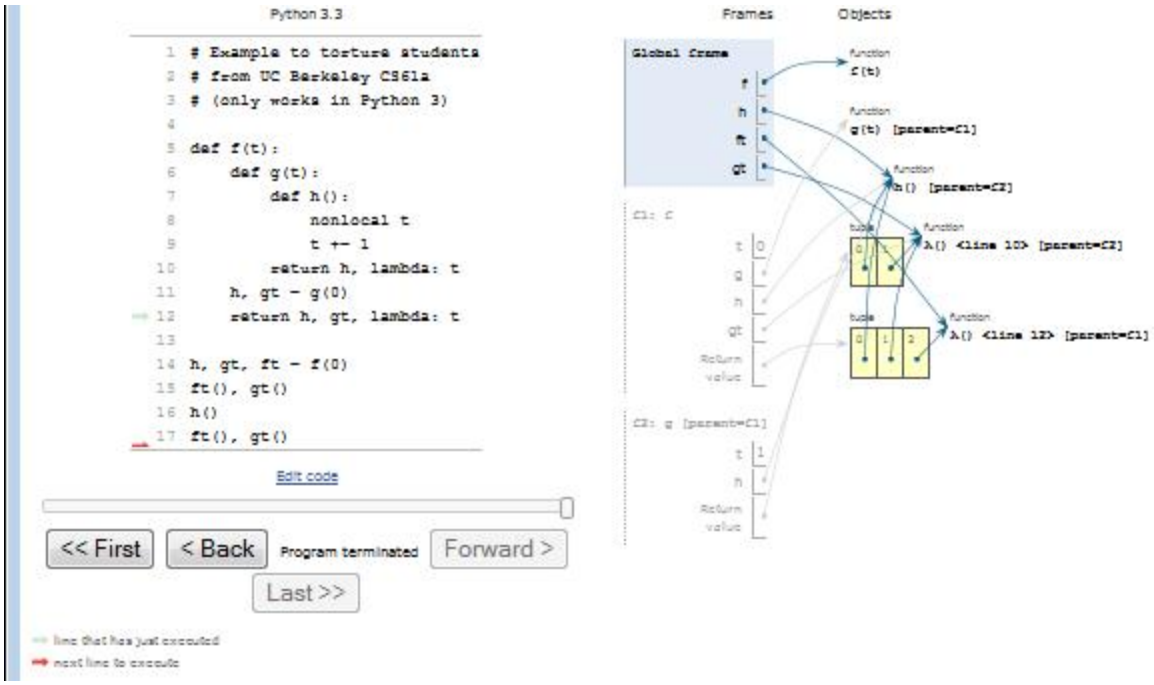
If you are in a hurry, you can press the **First** button and the **Last** button to cause the code visualizer to go the first or last instruction in one step.

The student torture program

There are links to quite a few sample programs below the drop-down boxes shown in Figure 1 . *(I cropped them out of Figure 1 to save space.)* One of those sample programs is named **student torture** .

As a preview of things to come, Figure 5 shows the result of opening that program in the code visualizer and pressing the **Last** button.

Figure 5. The student torture program.



Sharing the code visualizer with a tutor

Finally, there is another interesting sample program at http://pythontutor.com/ that you can step through and observe its behavior. While you are there, be sure to note your ability to share the code visualizer with a friend or a tutor on another computer on the Internet and jointly analyze a program.

Run the programs

I encourage you to use the code in Listing 1 and Listing 2. along with the code visualizer to repeat the procedures explained in this module. Confirm that you get results similar to those shown in Figure 3 and Figure 4 .

Change the options in the drop-down boxes and observe the results.

Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

As mentioned earlier, I strongly recommend that from this point forward in this course, you copy and paste every sample program into the code visualizer and observe the behavior of that code as you step through the program, one instruction at a time. I believe that doing that will help you to better understand Python programming.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1065-Visualizing Python
- File: Itse1359-1065.htm
- Published: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1080r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1080-Lists Part 1.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 , 17 , 18
- Listing index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1080-Lists Part 1* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? A *subscription* in Python is how you go about getting magazines delivered to your mailbox.

Go to answer 1

Question 2

Name three types of *sequence* objects.

Go to answer 2

Question 3

Given the following nomenclature

primary "[" expression_list "]"

what is the requirement for the *primary* ?

Go to answer 3

Question 4

If the *primary* is a *sequence* , what must be the type of expression_list?

Go to answer 4

Question 5

True or False? If the primary is a *sequence* , the expression_list must evaluate to a positive integer or to a slice containing positive integers.

Go to answer 5

Question 6

Which item in the sequence is selected for an index value of -1?

Go to answer 6

Question 7

True or False? Just like C, C++, and Java, Python supports a *character* type.

Go to answer 7

Question 8

What is another name for a *mapping* ?

Go to answer 8

Question 9

What is a *mutable* sequence?

Go to answer 9

Question 10

True or False? A string is a sequence.

Go to answer 10

Question 11

True or False? A string is a mutable sequence.

Go to answer 11

Question 12

What kinds of items can be placed in a *list* ?

Go to answer 12

Question 13

How are *lists* formed from a syntax viewpoint?

Go to answer 13

Question 14

Show how to create a simple list using program code.

Go to answer 14

Question 15

Show how to access an item in a list using a subscription.

Go to answer 15

Question 16

Show how to access a slice from a list.

Go to answer 16

Question 17

Show how to concatenate two lists.

Go to answer 17

Question 18

Show how to modify an item in a list using a subscription.

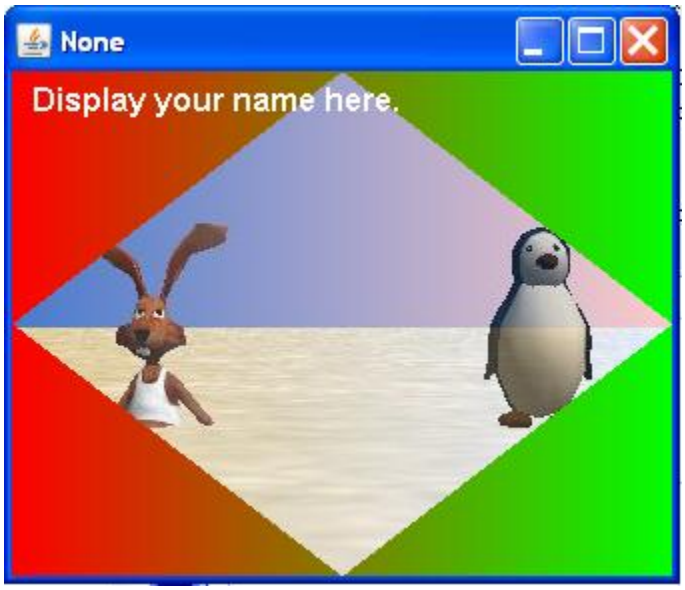
Go to answer 18

Listing index

- Listing 1 . Answer 14.
- Listing 2 . Answer 15.
- Listing 3 . Answer 16.
- Listing 4 . Answer 17.
- Listing 5 . Answer 18.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 18

Listing 5 shows how to modify an item in a list using a subscription.

Listing 5 . Answer 18.

```
listA = [3.14,59,"A string",1024]
listA[2] = "New string"
```

Go back to Question 18

Answer 17

Listing 4 shows how to concatenate two lists.

Listing 4 . Answer 17.

```
listA = [3.14,59,"A string",1024]
listB = [2,4,6,16]
listC = listA + listB
```

Go back to Question 17

Answer 16

Listing 3 shows how to access a slice from a list.

Listing 3 . Answer 16.

```
print(theList[0:3])
```

Go back to Question 16

Answer 15

Listing 2 shows how to access an item in a list using a subscription.

Listing 2 . Answer 15.

Listing 2 . Answer 15.

```
print(theList[2])
```

Go back to Question 15

Answer 14

Listing 1 shows a simple list.

Listing 1 . Answer 14.

```
theList = [3.14,59,"A string",1024]
```

Go back to Question 14

Answer 13

Lists are formed by placing a comma-separated sequence of expressions in square brackets.

Go back to Question 13

Answer 12

The items of a *list* are arbitrary Python objects.

Go back to Question 12

Answer 11

False. The characters in a string cannot be modified after the string is created.

Go back to Question 11

Answer 10

True. A string is a sequence.

Go back to Question 10

Answer 9

Mutable sequences can be changed after they are created.

Go back to Question 9

Answer 8

A dictionary.

Go back to Question 8

Answer 7

False. There is no character type in Python. Rather, a string's items are characters. A character is not a separate data type but a string containing exactly one character.

Go back to Question 7

Answer 6

The last item in the sequence is selected for an index value of -1.

Go back to Question 6

Answer 5

False. The integers may be positive or negative.

Go back to Question 5

Answer 4

If the primary is a *sequence* , the expression_list must evaluate to an integer or a slice.

Go back to Question 4

Answer 3

The *primary* must evaluate to an object of a *sequence* or *mapping* type.

Go back to Question 3

Answer 2

string, tuple and list and possibly range object

Go back to Question 2

Answer 1

False in the Python context. In Python, A *subscription* selects an item of a *sequence* object.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1080r-Review
- File: Itse1359-1080r.htm
- Published: 10/15/14
- Revised: 02/21/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1090-Lists Part 2
This module expands on the previous module on lists by teaching you other ways to manipulate lists.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Preface
- Introduction
    - Manipulating lists
    - Other ways to manipulate lists
- Some sample programs
    - Replacing a slice
        - Can change the length of the list
        - The function named len()
        - Replaces a three-element slice with a six-element list
        - Length of the list is increased by 3
        - Program output
    - Replacing an element with a list
        - Behavior is different from above
        - Produces nested lists
        - Length is unchanged
        - Program output from replacing an element with a list
        - One element is itself a list
    - A visualization

    - Extracting elements from a nested list
        - Display the nested list combination
        - Double-square-bracket notation
        - Program output from extracting elements from a nested list
    - More on nested elements
        - Create a list of lists
        - Inner lists are different lengths
        - Program behavior and output
        - Getting the length of a nested list
    - Arrays -- not for beginners
        - Python lists are more powerful
        - Sub-arrays can be different sizes
        - Types can also be different
    - A three-dimensional array program
        - Triple-square-bracket notation
        - Visualization of a three-dimensional array
        - Program behavior
        - Output from a three-dimensional array program
- More information on lists
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. It expands on the previous module titled Itse1359-1080-Lists Part 1 by teaching you some other ways to manipulate lists.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Program output.
- Figure 2 . Program output from replacing an element with a list.
- Figure 3 . Visualization
- Figure 4 . Program output from extracting elements from a nested list.
- Figure 5 . Output from more nested elements.
- Figure 6 . Visualization of a three-dimensional array.
- Figure 7 . Output from a three-dimensional array program.

Listings

- Listing 1 . Replace a slice in a list.
- Listing 2 . Replacing an element with a list.
- Listing 3 . Extracting elements from a nested list.
- Listing 4 . More nested elements.
- Listing 5 . A three-dimensional array program.

Introduction

A previous module titled Itse1359-1080-Lists Part 1 introduced you to lists and some other structures as well.

It also introduced you to

- subscriptions
- sequences
- mutable sequences
- mappings
- slicing, and
- tuples.

Manipulating lists

That module showed you some of the ways that you can manipulate lists. The discussion was illustrated using sample programs and visualizations.

Other ways to manipulate lists

This module carries that discussion forward by using sample programs to teach you other ways to manipulate lists.

Some sample programs

Replacing a slice

You can replace a slice in a list with the elements from another list through assignment.

Can change the length of the list

Note that this operation can change the length of the original list.

The script in Listing 1 replaces a slice in a list with the elements from another list.

Listing 1 . Replace a slice in a list.

```
# Illustrates replacing a slice in a list
#
#----------------------------------------
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace a slice")
listA[1:4] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
```

The function named len()

This program also illustrates the use of the function named **len()** to get and print the length of the list.

Replaces a three-element slice with a six-element list

In this program, a slice of an original five-element list, consisting of the elements from 1 through 3 inclusive, is replaced by the elements of a new list consisting of six new elements.

Length of the list is increased by 3

Since three existing elements are replaced by six new elements, the overall length of the list is increased by three elements.

Program output

The output from this program is shown in Figure 1 .

Figure 1 . Program output.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace a slice
Print the modified list
[100, 2, 4, 8, 16, 32, 64, 500]
Modified length is:
8
```

As you can see in Figure 1 , the six new elements replaced the three original elements to increase the length of the list from 5 to 8 elements.

Replacing an element with a list

It is also possible to replace an element in an existing list with a new list, as illustrated by the program in Listing 2 .

Behavior is different from above

In this case, the behavior is different from that shown above where a slice from the original list was replaced with the elements from a different list *(even though the right operand of the assignment operator is the same in both cases)* .

Produces nested lists

When a single element is replaced by a list, the result is that *a new list is nested inside the original list* .

Length is unchanged

It is also interesting to note that the length of the list is unchanged by this operation since the list that replaces the element is itself considered to be a single element. Therefore, the number of elements is not changed.

This is illustrated in Listing 2 , where the element at index 2 of an original list is replaced with a new list having six elements.

Listing 2 . Replacing an element with a list.

Listing 2 . Replacing an element with a list.

```
# Illustrates replacing an
# element with a slice
#
#-------------------------------
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace an element")
listA[2] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
```

Program output from replacing an element with a list

The output from this program is shown in Figure 2 .

Figure 2 . Program output from replacing an element with a list.

Figure 2 . Program output from replacing an element with a list.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
```

One element is itself a list

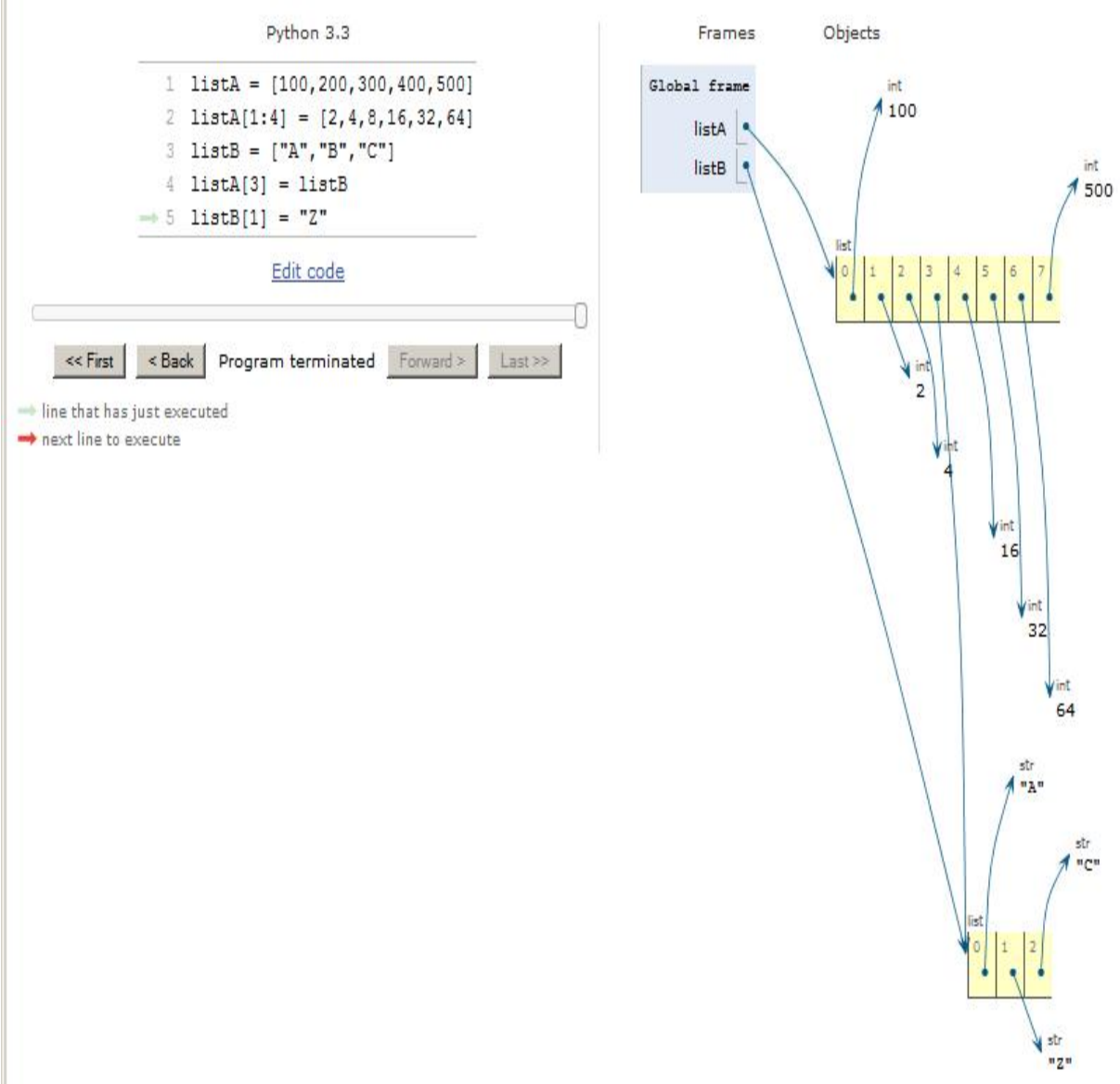
As you can see from Figure 2 , the result is that one of the elements in the original five-element list is replaced by a new list containing six elements. However, the length of the list is unchanged.

Again, it is important to note that this results in one list being nested inside of another list.

A visualization

The code visualization shown in Figure 3 summarizes the concepts that were explained in the previous two sections. This image was produced by entering the five statements shown in the code visualizer's code window and stepping through all five instructions.

Figure 3. Visualization.



I recommend that you use the code visualizer to step through this code and observe the state of the program in memory as each instruction is executed. Try to correlate the behavior of this program with what you learned in the sections above titled Replacing a slice and Replacing an element with a list .

Extracting elements from a nested list

Now I am going to illustrate the syntax for extracting elements from a nested list using *pairs of matching square brackets* . Listing 3 is an expansion of Listing 2 .

Listing 3 . Extracting elements from a nested list.

```
# Illustrates extracting a
# list element and extracting
# elements from a nested list
#
#-------------------------------
print("Create and print a list")
listA = [100,200,300,400,500]
print(listA)
print("Original length is:")
print(len(listA))
print("Replace an element")
listA[2] = [2,4,8,16,32,64]
print("Print the modified list")
print(listA)
print("Modified length is:")
print(len(listA))
print("Extract and display each")
print(" element in the list")
print(listA[0])
print(listA[1])
print(listA[2])
print(listA[3])
print(listA[4])
```

Listing 3 . Extracting elements from a nested list.

```
print("Extract and display each")
print(" element in nested list")
print(listA[2][0])
print(listA[2][1])
print(listA[2][2])
print(listA[2][3])
print(listA[2][4])
print(listA[2][5])
```

Display the nested list combination

After nesting a list as element 2 in another list, the program displays the value of each of the elements of the original list. When element 2 is displayed, it can be seen to be another list.

Double-square-bracket notation

Then the program uses double-square-bracket notation ***(listA[2][4])*** to extract and display each of the elements in the nested inner list that comprises element 2 of the outer list.

Program output from extracting elements from a nested list

The output from this program is shown in Figure 4 .

Figure 4 . Program output from extracting elements from a nested list.

```
Create and print a list
[100, 200, 300, 400, 500]
Original length is:
5
Replace an element
Print the modified list
[100, 200, [2, 4, 8, 16, 32, 64], 400, 500]
Modified length is:
5
Extract and display each
element in the list
100
200
[2, 4, 8, 16, 32, 64]
400
500
Extract and display each
element in nested list
2
4
8
16
32
64
```

More on nested elements

The program in Listing 4 illustrates some additional aspects of nested elements.

Listing 4 . More nested elements.

```
# Illustrates more nested elements
#
#-----------------------------------------
print("Create and print a list\
with 3 nested elements")
listA = [[2,4],[8,16,32],[64,128,256,512]]
print(listA)
print("Number of elements is:")
print(len(listA))
print("Length of Element 0 is")
print(len(listA[0]))
print("Length of Element 1 is")
print(len(listA[1]))
print("Length of Element 2 is")
print(len(listA[2]))
```

Create a list of lists

This program defines a three-element list containing three nested lists.

In other words, each of the elements in the outer list is itself a list.

Inner lists are different lengths

Furthermore, the lengths of the inner nested lists are not the same. The lengths of the inner nested lists are 2, 3, and 4 elements each, respectively.

Program behavior and output

The output from the program is shown in Figure 5 . The program displays the entire list, and then gets and displays the lengths of each of the nested lists.

Figure 5 . Output from more nested elements.

```
Create and print a list with 3 nested elements
[[2, 4], [8, 16, 32], [64, 128, 256, 512]]
Number of elements is:
3
Length of Element 0 is
2
Length of Element 1 is
3
Length of Element 2 is
4
```

Getting the length of a nested list

Note in particular the syntax used to pass a parameter to the **len()** method in order to get the length of a nested list ***(len(listA[1]))*** .

Arrays -- not for beginners

If you are a beginning programmer, you may want to skip this section. If you are an experienced programmer, you may have observed that a Python

lists bear a striking resemblance to arrays in other programming environments.

Python lists are more powerful

However, Python lists are more powerful than the arrays I am aware of in other programming environments, including Java.

Sub-arrays can be different sizes

For example as in Java, when a Python list is used to construct a multidimensional array, the sub arrays don't have to be of the same size.

Types can also be different

However, unlike Java, the elements in the array don't even have to be of the same type *(granted that the elements in a Java array can be of different types so long as there is an inheritance or Interface relationship among them).*

A three-dimensional array program

The program in Listing 5 might represent what an experienced programmer would consider to be a three-dimensional array of integer data in some other programming environment.

Listing 5 . A three-dimensional array program.

Listing 5 . A three-dimensional array program.

```
# Illustrates a three-dimensional array list
#
#-----------------------------------------------
-------
print("Create and print a\
three-dimensional array list")
listA = [[[[1],[2]],[[3],[4]]],[[[5],[6]],
[[7],[8]]]]
print(listA)
print("Print each element")
print(listA[0][0][0])
print(listA[0][0][1])
print(listA[0][1][0])
print(listA[0][1][1])
print(listA[1][0][0])
print(listA[1][0][1])
print(listA[1][1][0])
print(listA[1][1][1])
```

Triple-square-bracket notation

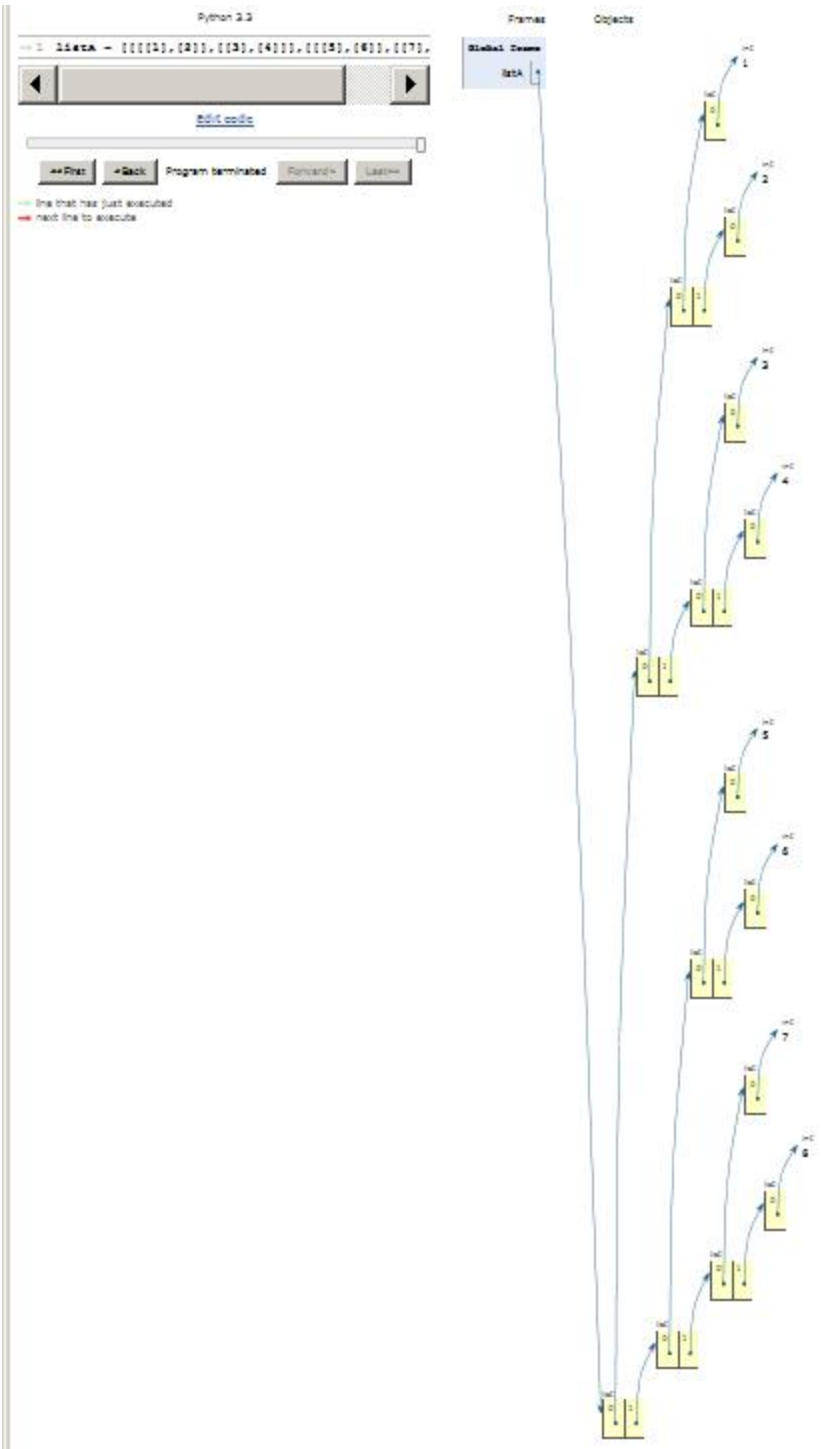
Pay particular attention to the triple square bracket notation that is used to access and print each element in the array. This is very similar to the syntax in other programming languages.

Visualization of a three-dimensional array

Figure 6 shows a visualization of the three-dimensional array pointed to by the variable named **listA** in Listing 5 . In programming, we deal with arrays

having one, two, three or more dimensions. As you can see, even a three-dimensional array is a relatively complex structure.

Figure 6. Visualization of a three-dimensional array.



Program behavior

This program

- Creates and populates the list that represents a three-dimensional array.
- Displays the entire array as a set of nested lists.
- Displays the contents of each element in the array.

Output from a three-dimensional array program

The output from the program is shown in Figure 7 . The triple-square-bracket notation in the program of Listing 5 is essentially the same notation that would be used to access the individual elements in a three-dimensional array in C, C++, or Java.

Figure 7 . Output from a three-dimensional array program.

Figure 7 . Output from a three-dimensional array program.

```
Create and print a three-dimensional array
list
[[[[1], [2]], [[3], [4]]], [[[5], [6]], [[7],
[8]]]]
Print each element
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
```

More information on lists

There is quite a lot more that you will need to learn about lists. However, I will defer that discussion for a future module where I discuss the use of a list as a *data structure* or a *container* .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1090-Lists Part 2
- File: Itse1359-1090.htm
- Published: 10/15/14

- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1110-Nested Tuples
This module will expand your knowledge of tuples by teaching you about nesting tuples within other tuples.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- What is a tuple?
- Sample program
    - Create two tuples
    - Nesting the tuples
    - Get the length of the new tuple
- Complete program listing
- Visualization of the tuples
- Run the program
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the

Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Output from the code in Listing 1.
- Figure 2 . Output from the code in Listing 2.
- Figure 3 . Output from the code in Listing 3.
- Figure 4 . Output from the code in Listing 4.
- Figure 5 . Visualization of the tuples

Listings

- Listing 1 . Beginning of the program.
- Listing 2 . Nesting the tuples.
- Listing 3 . Get the length of the new tuple.
- Listing 4 . Complete program listing.

Introduction

Previous modules have introduced you to lists, subscriptions, sequences, mutable sequences, mappings, slicings, and tuples.

The earlier module titled Itse1359-1100-Indexing and Slicing Tuples showed you:

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.

This module will expand your knowledge of tuples by teaching you about nesting tuples within other tuples.

What is a tuple ?

To briefly repeat part of what you learned in the earlier module, a tuple is like a list whose values cannot be modified. In other words, a tuple is immutable.

- Tuples are normally written as a sequence of items contained in *(optional)* matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.

Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested inside of other tuples.

Sample program

Listing 4 shows a complete listing of a Python program that

- Creates two tuples.
- Nests them in a third tuple.
- Determines the length *(number of items)* in the tuple containing the two nested tuples.
- Prints various items of information along the way.

A visualization of the tuples in this program is provided in Figure 5 .

Create two tuples

The beginning of the program is shown in Listing 1 .

Listing 1 . Beginning of the program.

```
# Illustrates creating and displaying nested
tuples
#-----------------------------------------------
-----

print("Create/print one tuple")
t1 = 1,2
print(t1)
print("Create/print another tuple")
t2 = "a","b"
print(t2)
```

The code in Listing 1 creates and displays the two tuples.

Figure 1 shows the output produced by the code fragment in Listing 1 .

Figure 1 . Output from the code in Listing 1.

```
Create/print one tuple
(1, 2)
Create/print another tuple
('a', 'b')
```

The remaining parts of this program will be presented and discussed as code fragments in subsequent Listings. As mentioned above, a complete listing of the program is shown in Listing 4 .

A consolidated view of the output from the program is shown in Figure 4 .

Nesting the tuples

The code in Listing 2 nests the two tuples, **t1** and **t2** , produced earlier, along with two strings, in a new tuple. The new tuple is assigned to the variable named **t3** .

Listing 2 . Nesting the tuples.

```
print("Create/print nested tuple")
t3 = "A",t1,"B",t2
print(t3)
```

All that is required to nest the existing tuples in a new tuple is to list the variables that reference the two existing tuples in a comma-separated list of items for creation of the new tuple. *(Note that the optional parentheses were omitted in Listing 2 .)*

Figure 2 shows the output for the new tuple containing two nested tuples.

Figure 2 . Output from the code in Listing 2.

```
Create/print nested tuple
('A', (1, 2), 'B', ('a', 'b'))
```

Note that the two nested tuples retain their identity as tuples, as indicated by the fact that the parentheses surrounding the items in the two nested tuples are preserved in the new tuple. This is also indicated by the visualization in Figure 5 .

Get the length of the new tuple

The code in Listing 3 shows code that gets and displays the length of the new tuple that contains the two nested tuples.

Listing 3 . Get the length of the new tuple.

```
print("Length of nested tuple is:")
print(len(t3))
```

The length is a measure of the number of items in the tuple, and is obtained using the method named **len** .

Figure 3 shows the output produced by the code in Listing 3 , including the length of the new tuple containing the two nested tuples.

Figure 3 . Output from the code in Listing 3.

```
Length of nested tuple is:
4
```

It is important to note that even though the tuple shown in Figure 2 actually consists of six individual items *(ignoring parentheses)* , each of the nested tuples is treated as a single item, giving a length of only four items for the tuple that contains the two nested tuples.

This would be true regardless of the length of the nested tuples.

You will learn in a future module that a double square-bracket indexing notation can be used to gain access to the individual items in tuples that are nested inside of other tuples.

Complete program listing

A complete listing of the program is shown in Listing 4 .

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates creating and displaying nested
tuples
#-----------------------------------------------
-----

print("Create/print one tuple")
t1 = 1,2
print(t1)
print("Create/print another tuple")
t2 = "a","b"
print(t2)
print("Create/print nested tuple")
t3 = "A",t1,"B",t2
print(t3)
print("Length of nested tuple is:")
print(len(t3))
```

Figure 4 shows a consolidated view of the output produced by this program.

Figure 4 . Output from the code in Listing 4.

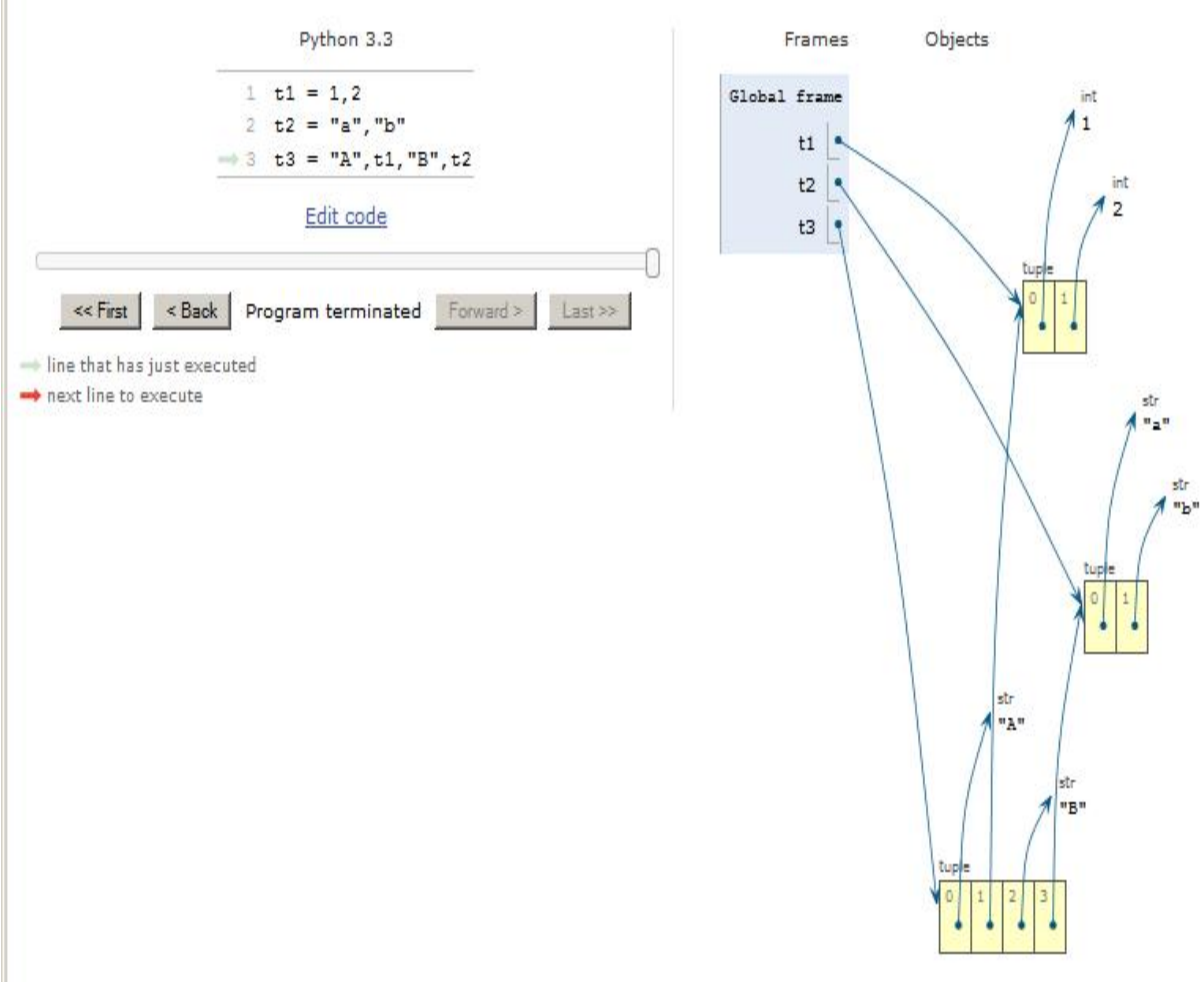
Figure 4 . Output from the code in Listing 4.

```
Create/print one tuple
(1, 2)
Create/print another tuple
('a', 'b')
Create/print nested tuple
('A', (1, 2), 'B', ('a', 'b'))
Length of nested tuple is:
4
```

Visualization of the tuples

A visualization of the tuples from this program is shown in Figure 5 .

Figure 5. Visualization of the tuples.



Run the program

I encourage you to copy the code from Listing 4 . Execute the code and confirm that you get the same results as those shown in Figure 4 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1110-Nested Tuples
- File: Itse1359-1110.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1110r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1110-Nested Tuples.

Table of contents

- Preface
- Questions
    - 1 , 2
- Figure index
- Listing index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1110-Nested Tuples* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? The program code in Listing 1 produces the output shown in Figure 1 .

Listing 1 . Question 1 program code.

```
t1 = 1,2
print(t1)

t2 = "a","b"
print(t2)

t3 = "A",t1,"B",t2
print(t3)
```

Figure 1 . Question 1 possible output.

Figure 1 . Question 1 possible output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
```

Go to answer 1

Question 2

True or False? The program code in Listing 2 produces the output shown in Figure 2 .

Listing 2 . Question 2 program code.

```
t1 = 1,2
print(t1)

t2 = "a","b"
print(t2)

t3 = "A",t1,"B",t2
print(t3)
print(len(t3))
```

Figure 2 . Question 2 possible output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
6
```

Go to answer 2

Figure index

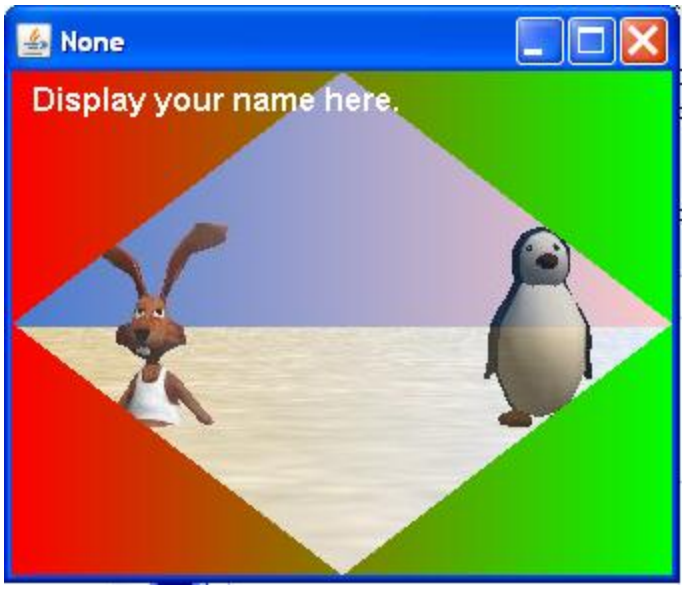
- Figure 1 . Question 1 possible output.
- Figure 2 . Question 2 possible output.
- Figure 3 . Question 2 actual output.

Listing index

- Listing 1 . Question 1 program code.
- Listing 2 . Question 2 program code.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 2

False. The actual output is shown in Figure 3 . The code in Listing 2 produces a tuple containing two other nested tuples. A nested tuple counts

as only one element in the length of a tuple containing nested tuples.

Figure 3 . Question 2 actual output.

```
(1, 2)
('a', 'b')
('A', (1, 2), 'B', ('a', 'b'))
4
```

Go back to Question 2

Answer 1

True.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1110r-Review
- File: Itse1359-1110r.htm
- Published: 10/19/14

- Revised: 02/23/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1120-Empty and Single-Item Tuples
This module will teach you how to create empty tuples and tuples containing only one item.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- What is a tuple?
- Sample program
    - Empty, single-item, and nested tuples
    - An empty tuple
    - A tuple with only one element
    - Nested tuples
    - Doubly-nested tuples
- Complete program listing
- Visualization of the tuples
- Run the program
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Output from the code in Listing 1.
- Figure 2 . Output from the code in Listing 2.
- Figure 3 . Output from the code in Listing 3.
- Figure 4 . Consolidated output from the program in Listing 4.
- Figure 5 . Visualization of the tuples.

Listings

- Listing 1 . Beginning of the program.
- Listing 2 . A tuple with only one element.
- Listing 3 . Create and print nested tuples.
- Listing 4 . Create and print nested tuples.

Introduction

This module is part of a series of modules designed to teach you about tuples in Python.

Previous modules have illustrated

- How to create a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.

This module will teach you how to create empty tuples and tuples containing only one item.

What is a tuple ?

A tuple is like a list whose values cannot be modified. It is an ordered list of objects, and it can contain references to any type of object.

- Tuples are normally written as a sequence of items contained in *(optional)* matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.
- Tuples can be nested.

Sample Program

Empty, single-item, and nested tuples

Listing 4 shows a Python program that

- Creates an empty tuple.
- Creates a single-item tuple.
- Nests the two tuples along with some strings in a third tuple.
- Determines the length of each of the tuples.
- Displays all of the above

A consolidated output from the program is shown in Figure 4 .

A visualization of the tuples in the program is provided in Figure 5 .

An empty tuple

I will explain this program in fragments. The beginning of the program is shown in Listing 1 .

Listing 1 . Beginning of the program.

```
# Illustrates empty tuples and tuples with
only one element
#-----------------------------------------
-------------

print("Create and print empty tuple")
t1 = ()
print(t1)
print("Length of empty tuple is")
print(len(t1))
```

As you might have guessed from the name, an empty tuple is just a pair of empty parentheses as shown by the second statement in Listing 1 . *(The parentheses aren't optional for an empty tuple.)*

Figure 1 shows the output produced by the code in Listing 1 . The empty tuple is displayed simply as a pair of empty parentheses, and the length of the empty tuple is shown to be zero.

Figure 1 . Output from the code in Listing 1.

Figure 1 . Output from the code in Listing 1.

```
Create and print empty tuple
()
Length of empty tuple is
0
```

A tuple with only one element

There are probably no surprises regarding an empty tuple. However, there may be some surprises in the code fragment shown in Listing 2 . This fragment deals with a tuple containing only one element.

Listing 2 . A tuple with only one element.

```
print("Create and print one-element tuple")
# Note the req trailing comma
t2 = "a",
print(t2)
print("Length of one-element tuple is:")
print(len(t2))
```

The syntax for creating a tuple with only one element is rather ugly, but is required to avoid ambiguity. In particular, it is necessary to follow the

single tuple item with a comma as shown in the third line of text in Listing 2 .

Had I written that line simply as follows without the extra comma,

t2 = "a"

the result would have been to create a new variable named **t2** whose contents would be the string "a". *(The parentheses are optional here but the comma is required with or without the parentheses.)*

This would not indicate a tuple at all. The extra comma is required to make a single-item tuple unique and to distinguish it from other possibilities.

Figure 2 shows the output produced by the code in Listing 2 . The single-item tuple is shown in the third line of text in Listing 2 . As is always the case, the tuple is displayed in parentheses.

Figure 2 . Output from the code in Listing 2.

```
Create and print one-element tuple
('a',)
Length of one-element tuple is:
1
```

The length of the tuple as shown in Figure 2 is one (1) item.

Nested tuples

Just to give you a little more practice in dealing with nested tuples, the code in Listing 3 nests the two tuples created above into a new tuple and stores a reference to the new tuple in the variable named **t3** .

Listing 3 . Create and print nested tuples.

```
print("Create and print nested tuple")
t3 = "A",t1,"B",(t2,"Z"),"C"
print(t3)

print("Length of nested tuple is")
print(len(t3))
```

Doubly-nested tuples

However unlike previous sample programs, in this case, literal parentheses are used to cause the tuple named **t2** to be doubly nested.

In particular, as shown by the second statement in Listing 3 , the tuple named **t2** and the string **"Z"** are used to create a tuple, which in turn, is nested in the tuple assigned to the variable named **t3** . This is also shown in the visualization in Figure 5 .

The double nesting is evidenced by the extra parentheses in the second line of text in the output shown in Figure 3 .

Figure 3 . Output from the code in Listing 3.

```
Create and print nested tuple
('A', (), 'B', (('a',), 'Z'), 'C')
Length of nested tuple is
5
```

The length of the tuple is also shown in Figure 3 . Even though the tuple named **t3** contains two nested tuples *(one of which is doubly-nested)* , its overall length is only five (5) items.

One of the tuples nested inside of **t3** has a length of zero but it still counts as one item when the length of **t3** is determined.

Complete program listing

A complete listing of the program is shown in Listing 4 .

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates empty tuples and tuples with
only one element
#-----------------------------------------
------------

print("Create and print empty tuple")
t1 = ()
print(t1)
print("Length of empty tuple is")
print(len(t1))

print("Create and print one-element tuple")
# Note the req trailing comma
t2 = "a",
print(t2)
print("Length of one-element tuple is:")
print(len(t2))


print("Create and print nested tuple")
t3 = "A",t1,"B",(t2,"Z"),"C"
print(t3)

print("Length of nested tuple is")
print(len(t3))
```

A consolidated output from the program is shown in Figure 4 .

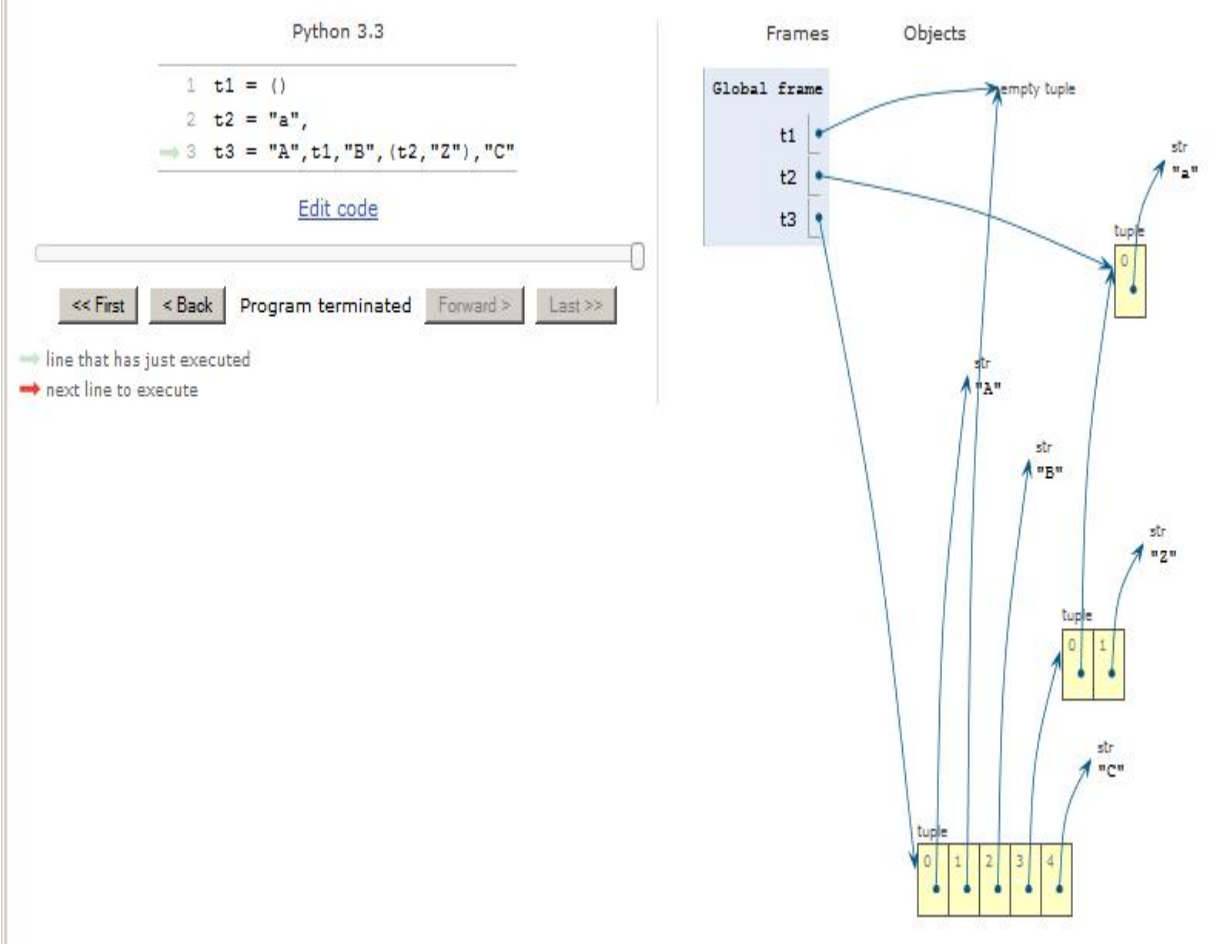
Figure 4 . Consolidated output from the program in Listing 4.

```
Create and print empty tuple
()
Length of empty tuple is
0
Create and print one-element tuple
('a',)
Length of one-element tuple is:
1
Create and print nested tuple
('A', (), 'B', (('a',), 'Z'), 'C')
Length of nested tuple is
5
```

Visualization of the tuples

A visualization of the tuples in the program is provided in Figure 5 .

Figure 5. Visualization of the tuples.



Run the program

I encourage you to copy the code from Listing 4 . Execute the code and confirm that you get the same results as those shown in Figure 4 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1120-Empty and Single-Item Tuples
- File: Itse1359-1120.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1120r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1120-Empty and Single-Item Tuples.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1120-Empty and Single-Item Tuples* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Questions

Question 1

True or False? The code in Figure 1 creates and prints an empty tuple producing the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
t1 =
print(t1)

print(len(t1))
```

Figure 2 . Question 1 possible output.

```
()
0
```

Go to answer 1

Question 2

True or False? The code in Figure 4 creates and prints an empty tuple producing the output shown in Figure 5 .

Figure 4 . Question 2 program code.

```
t1 = ()
print(t1)

print(len(t1))
```

Figure 5 . Question 2 possible output.

```
()
0
```

Go to answer 2

Question 3

True or False? The code in Figure 6 produces the output shown in Figure 7 .

Figure 6 . Question 3 program code.

```
t2 = ("a")
print(t2)
```

Figure 7 . Question 3 possible output.

```
('a')
```

Go to answer 3

Question 4

True or False? The code in Figure 9 produces the output shown in Figure 10 .

Figure 9 . Question 4 program code.

```
t2 = ("a",)
print(t2)
```

Figure 10 . Question 4 possible output.

```
('a',)
```

Go to answer 4

Question 5

True or False? The code in Figure 11 produces the output shown in Figure 12 .

Figure 11 . Question 5 program code.

Figure 11 . Question 5 program code.

```
t1 = ()
print(t1)
print(len(t1))

t2 = ("a",)
print(t2)
print(len(t2))

t3 = "A",t1,"B",(t2,"Z"),"C"
print(t3)
print(t3[3][0])
print(len(t3))
```

Figure 12 . Question 5 possible output.

```
()
0
('a',)
1
('A', (), 'B', (('a',), 'Z'), 'C')
('a',)
5
```

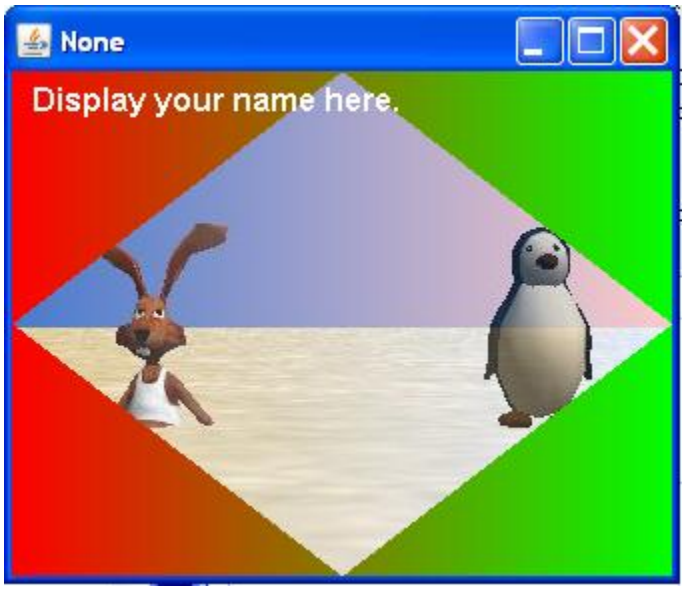
Go to answer 5

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.
- Figure 3 . Question 1 actual output.
- Figure 4 . Question 2 program code.
- Figure 5 . Question 2 possible output.
- Figure 6 . Question 3 program code.
- Figure 7 . Question 3 possible output.
- Figure 8 . Question 3 actual output.
- Figure 9 . Question 4 program code.
- Figure 10 . Question 4 possible output.
- Figure 11 . Question 5 program code.
- Figure 12 . Question 5 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 5

True. Note the indexing using double square bracket notation.

Go back to Question 5

Answer 4

True.

Go back to Question 4

Answer 3

False. The output is shown in Figure 8 . The code in Figure 6 doesn't recognize the item in parentheses as a tuple. It is being treated as an ordinary variable. It needs a comma to be recognized as a tuple.

Figure 8 . Question 3 actual output.

```
a
```

Go back to Question 3

Answer 2

True.

Go back to Question 2

Answer 1

False. The actual output is shown in Figure 3 . While the parentheses are optional when creating and populating a non-empty tuple, they are not optional when creating an empty tuple.

Figure 3 . Question 1 actual output.

```
   File "1359-1120r-01.py", line 3
    t1 =
        ^
SyntaxError: invalid syntax
```

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1120r-Review
- File: Itse1359-1120r.htm
- Published: 10/19/14
- Revised: 02/23/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1130-Unpacking Tuples
This module will teach you about unpacking tuples in Python.

Table of contents

- Preface
    - Viewing tip
        - Figures
        - Listings
- Introduction
- What is a tuple?
- Sample program
    - Program to unpack a tuple
    - Tuples can be concatenated
    - Unusual syntax
    - Packing and unpacking a tuple
    - Unpack the tuple into the mutable list
- Complete program listing
- Visualization of the tuples in the program
- Run the program
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display.)

Figures

- Figure 1 . Output from the code in Listing 1.
- Figure 2 . Output from the code in Listing 2.
- Figure 3 . Output from the code in Listing 3.
- Figure 4 . Output from the code in Listing 4.
- Figure 5 . Visualization of the tuples in the program.
- Figure 6 . Output from the modified program.

Listings

- Listing 1 . Beginning of the program.
- Listing 2 . Unpack the tuple and print individual elements.
- Listing 3 . Unpack the tuple into the mutable list.
- Listing 4 . Complete program listing.

Introduction

This module is part of a series of modules designed to teach you about tuples.

Earlier modules have illustrated

- How to create *(pack)* a tuple.
- How to access a tuple item using indexing.
- How to slice a tuple.
- How to nest tuples.

- How to create empty tuples.
- How to create single-item tuples.

This module will teach you about unpacking tuples.

What is a tuple ?

A tuple is an immutable ordered list of objects. It can contain references to any type of object. See earlier modules in this series for a more detailed description.

Sample program

Program to unpack a tuple

Listing 4 presents a Python program that:

- Creates (packs) two simple tuples.
- Creates (packs) a third tuple by concatenating the first two tuples.
- Displays the third tuple.
- Unpacks the third tuple, assigning each item of the tuple into a separate variable.
- Displays each of the variables.
- Creates and displays a mutable list object containing five strings.
- Unpacks the tuple created earlier assigning the four items from the tuple into the first four items in the list.
- Displays the list.

Figure 4 shows the output produced by the program in Listing 4 .

Figure 5 shows a visualization of the tuples in the program after the first five statements in the code block have been executed. This is a case where you need to step through the program and observe changes in the diagram on the right to appreciate the behavior of the program.

I will explain this program in fragments. Listing 1 shows the beginning of the program.

Listing 1 . Beginning of the program.

```
# Illustrates unpacking a tuple
#------------------------------

# Create a pair of tuples
t1 = 1,2
t2 = "A","B"

# Concatenate and print them
t3 = t1 + t2
print(t3)
```

Tuples can be concatenated

As shown in Listing 1 , tuples support the concatenation (+) operator. You can concatenate two or more tuples to produce a new tuple. This program creates two simple tuples, and then concatenates them to create a third tuple.

Figure 1 shows the output produced by the code in Listing 1 . By now, the creation and display of simple tuples should be very familiar to you based on earlier modules. Therefore, I won't discuss this part of the program further.

Figure 1 . Output from the code in Listing 1.

Figure 1 . Output from the code in Listing 1.

```
(1, 2, 'A', 'B')
```

Unusual syntax

The code in Listing 2 is not quite so straightforward. In fact, it looks rather strange if you come from a conventional C, C++, or Java programming background.

Listing 2 . Unpack the tuple and print individual elements.

```
# Unpack the tuple and print individual
elements
w,x,y,z = t3
print(w)
print(x)
print(y)
print(z)
```

Figure 2 shows the output produced by the code in Listing 2 .

Figure 2 . Output from the code in Listing 2.

```
1
2
A
B
```

If you compare this output with the original tuple in Listing 1 , or with the previous output in Figure 1 , you will see that each of the individual items in the tuple *(in left-to-right order)* were assigned respectively to the variables named **w** , **x** , **y** , and **z** .

Thus, the lines of output produced by printing these four variables in Figure 2 match the items in the original tuple that was created in Listing 1 and displayed in Figure 1 .

Packing and unpacking a tuple

Version 3 of The Python Tutorial -- 5.3. Tuples and Sequences refers to the first, second, and third statements in Listing 1 as tuple packing using the following example:

> The statement t = 12345, 54321, 'hello!' is an example of tuple packing: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible

An early version of The Python Tutorial -- 5.3 Tuples and Sequences refers to the first statement in Listing 2 as *tuple unpacking* .

The current version of The Python Tutorial -- 5.3. Tuples and Sequences *(September 2014)* refers to that operation more generally as *sequence unpacking* .

That tutorial continues by telling us

> "... sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence."

If you try to run a program that doesn't meet these criteria, you will get an error similar to the following:

ValueError: too many values to unpack (expected 3)

Unpack the tuple into the mutable list

Just to make things a little more interesting, I decided to combine the use of a tuple *(an immutable list)* and a regular mutable list in this program.

Listing 3 contains the code to create a mutable list populated with five string characters.

Listing 3 . Unpack the tuple into the mutable list.

Listing 3 . Unpack the tuple into the mutable list.

```
# Create and print a list
L1 = ["a","b","c","d","e"]

# Unpack tuple into the list and print it
print(L1)
L1[0],L1[1],L1[2],L1[3] = t3
print(L1)
```

Then the list is displayed, as shown by the first line of text in Figure 3 . The first line of output in Figure 3 shows the contents of the list just after it is created and populated.

Figure 3 . Output from the code in Listing 3.

```
['a', 'b', 'c', 'd', 'e']
[1, 2, 'A', 'B', 'e']
```

Then the code in Listing 3 unpacks the four-element tuple referred to by **t3** into the first four elements in the list. *(Remember, a list is mutable, so the values of its items can be changed.)*

Then the contents of the list are displayed again producing the second line of text in Figure 3 . As you can see from Figure 3 , the first four items in the

list were replaced by the four items from the tuple. The fifth item in the list was not modified.

Complete program listing

A complete listing of the program discussed above is provided in Listing 4 .

Listing 4 . Complete program listing.

Listing 4 . Complete program listing.

```
# Illustrates unpacking a tuple
#------------------------------

# Create a pair of tuples
t1 = 1,2
t2 = "A","B"

# Concatenate and print them
t3 = t1 + t2
print(t3)

# Unpack the tuple and print individual
elements
w,x,y,z = t3
print(w)
print(x)
print(y)
print(z)

# Create and print a list
L1 = ["a","b","c","d","e"]
# Unpack tuple into the list and print it
print(L1)
L1[0],L1[1],L1[2],L1[3] = t3
print(L1)
```

Figure 4 shows the output from the code in Listing 4 .

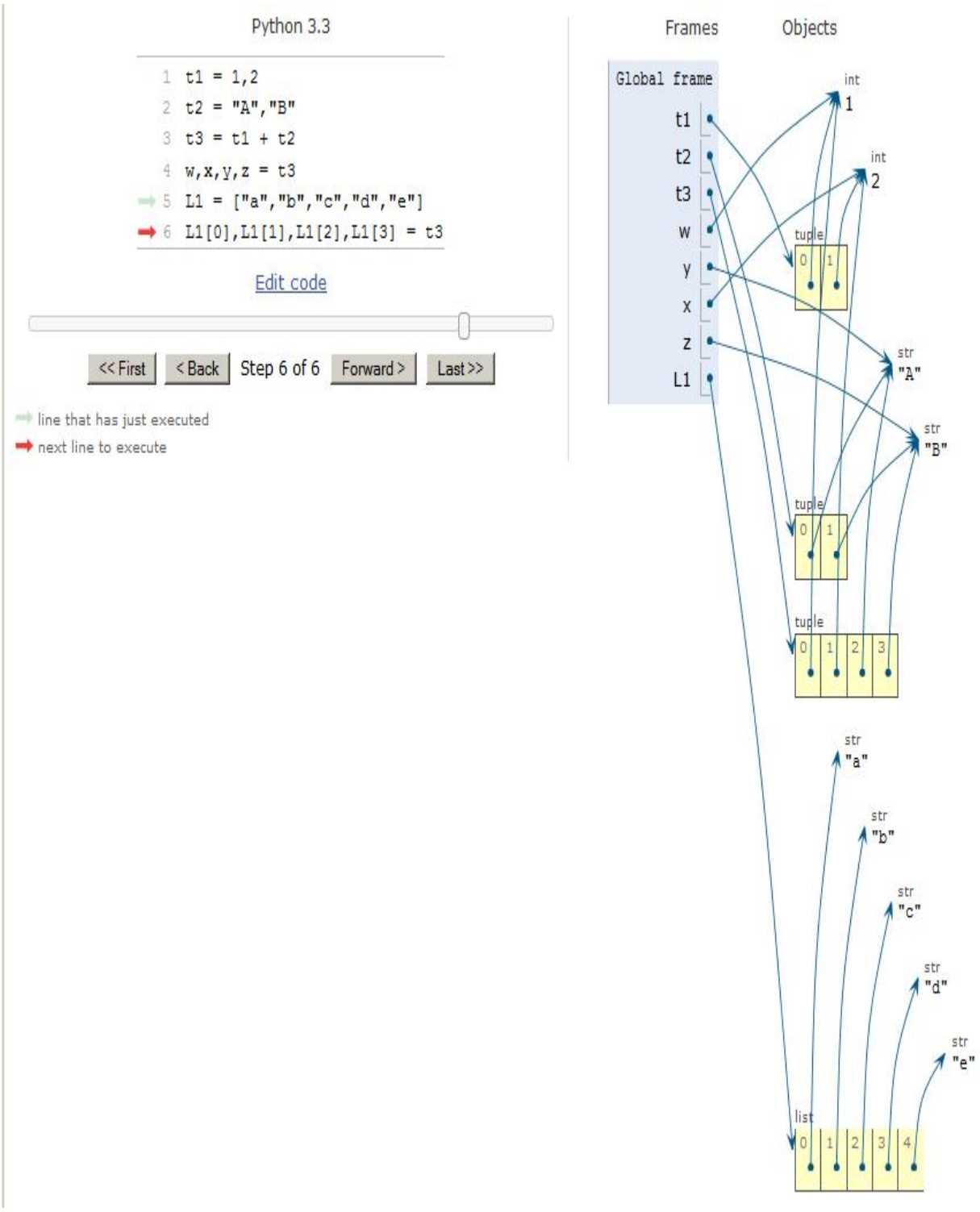
Figure 4 . Output from the code in Listing 4.

```
(1, 2, 'A', 'B')
1
2
A
B
['a', 'b', 'c', 'd', 'e']
[1, 2, 'A', 'B', 'e']
```

Visualization of the tuples in the program

Figure 5 shows a visualization of the tuples in the program after the first five statements in the code block have been executed. This is a case where you need to step through the program and observe changes in the diagram on the right to appreciate the behavior of the program.

Figure 5. Visualization of the tuples in the program.



Run the program

I encourage you to copy the code from Listing 4 . Execute the code and confirm that you get the same results as those shown in Figure 4 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

For example, you might want to try the following experiment. There is a line in the program shown in Listing 4 that reads as follows:

L1 = ["a","b","c","d","e"]

Modify the program, changing this line so that it reads as follows:

L1 = ("a","b","c","d","e")

or

L1 = "a","b","c","d","e"

By now you should recognize that either of these modifications will change L1 from an ordinary mutable list to a tuple *(an immutable list)* .

Now execute the modified program. Your output should look something like that shown in Figure 6 .

Figure 6 . Output from the modified program.

Figure 6 . Output from the modified program.

```
(1, 2, 'A', 'B')
1
2
A
B
('a', 'b', 'c', 'd', 'e')
Traceback (most recent call last):
  File "1359-1130-07.py", line 24, in <module>
    L1[0],L1[1],L1[2],L1[3] = t3
TypeError: 'tuple' object does not support
item assignment
```

Everything should work well until the attempt is made to unpack the tuple named **t3** and to assign the items from that tuple into the individual items of the new tuple named **L1** .

The items in a tuple are immutable, meaning that they cannot be changed. Therefore, the program crashes at this point with the error shown in Figure 6 .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1130-Unpacking Tuples
- File: Itse1359-1130.htm
- Published: 10/19/14
- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1130r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1130- Unpacking Tuples.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1230-The if Statement* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
dogs = ["Affenpinscher","Afgan Hound","Akita"]

if len(dogs) != 2:
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])
print("Length of dogs list = " +
str(len(dogs)))

if not(len(dogs) == 3):
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])
print(dogs)
```

Figure 2 . Question 1 possible output.

```
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
['Affenpinscher', 'Afgan Hound', 'Akita']
```

Go to answer 1

Question 2

True or False? The code in Figure 3 produces the output shown in Figure 4 .

Figure 3 . Question 2 program code.

Figure 3 . Question 2 program code.

```
weather = ["sunshine","rain"]

weatherToday = weather[1]

if weatherToday == "rain":
  print("It's raining, visit the museum.")
else:
  print("Sunshine, go to the beach.");

print("Vacation is over, go home.")
```

Figure 4 . Question 2 possible output.

```
Sunshine, go to the beach.
Vacation is over, go home.
```

Go to answer 2

Question 3

True or False? The code in Figure 6 produces the output shown in Figure 7 .

Figure 6 . Question 3 program code.

```
weather = ["sunshine","rain","snow"]

weatherToday = weather[2]

if weatherToday == "rain":
  print("It's raining, visit the museum.")
elif weatherToday == "sunshine":
  print("Sunshine, go to the beach.")
else:
  print("It's snowing, go skiing.")

print("Vacation is over, go home.")
```

Figure 7 . Question 3 possible output.

```
It's snowing, go skiing.
Vacation is over, go home.
```

Go to answer 3

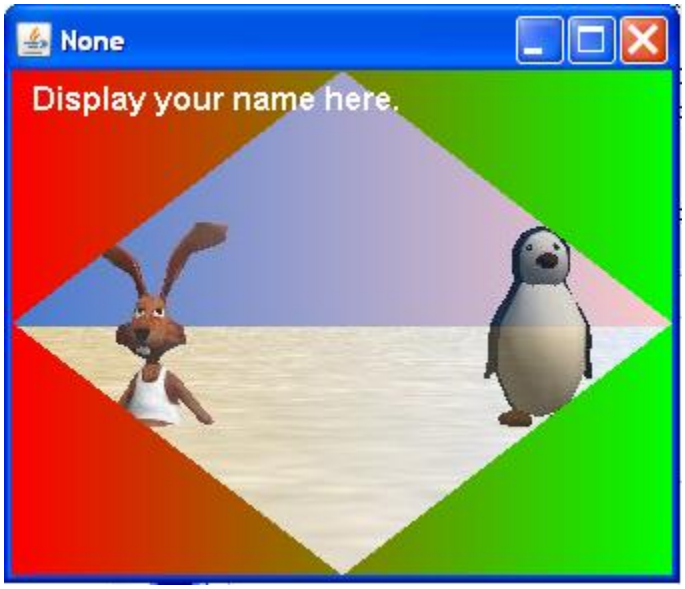
Figure index

- Figure 1 . Question 1 program code.

- Figure 2 . Question 1 possible output.
- Figure 3 . Question 2 program code.
- Figure 4 . Question 2 possible output.
- Figure 5 . Question 2 actual output.
- Figure 6 . Question 3 program code.
- Figure 7 . Question 3 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 3

True.

Go back to Question 3

Answer 2

False. The actual output is shown in Figure 5 .

Figure 5 . Question 2 actual output.

Figure 5 . Question 2 actual output.

```
It's raining, visit the museum.
Vacation is over, go home.
```

Go back to Question 2

Answer 1

True.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1230r-Review
- File: Itse1359-1230r.htm
- Published: 10/21/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1210r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1210-The While Loop.

Table of Contents

- Preface
- Questions
    - 1 , 2 , 3 , 4
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1210-The While Loop* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a","b","c"]
print(aList)

count = 6
while count <= 5:
  count = count + 1
  print(count)
  aList.append(count)

print(aList)
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
['a', 'b', 'c']
1
2
3
4
5
6
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to answer 1

Question 2

True or False? The code in Figure 4 produces the output shown in Figure 5 .

Figure 4 . Question 2 program code.

Figure 4 . Question 2 program code.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a","b","c"]
print(aList)

count = 0
while count <= 5:
  count = count + 1
print(count)
aList.append(count)

print(aList)
```

Figure 5 . Question 2 possible output.

Figure 5 . Question 2 possible output.

```
['a', 'b', 'c']
1
2
3
4
5
6
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to answer 2

Question 3

True or False? The code in Figure 7 produces the output shown in Figure 8 .

Figure 7 . Question 3 program code.

Figure 7 . Question 3 program code.

```
# Illustrates the use of a while loop to
manipulate a list

aList = ["a","b","c"]
print(aList)

count = 0
while count <= 5:
  count = count + 1
  print(count)
  aList.append(count)

print(aList)
```

Figure 8 . Question 3 possible output.

Figure 8 . Question 3 possible output.

```
['a', 'b', 'c']
1
2
3
4
5
6
['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

Go to answer 3

Question 4

True or False? The two statements shown in Figure 9 produce the same result.

Figure 9 . Question 9.

```
count = count + 1

count += 1
```

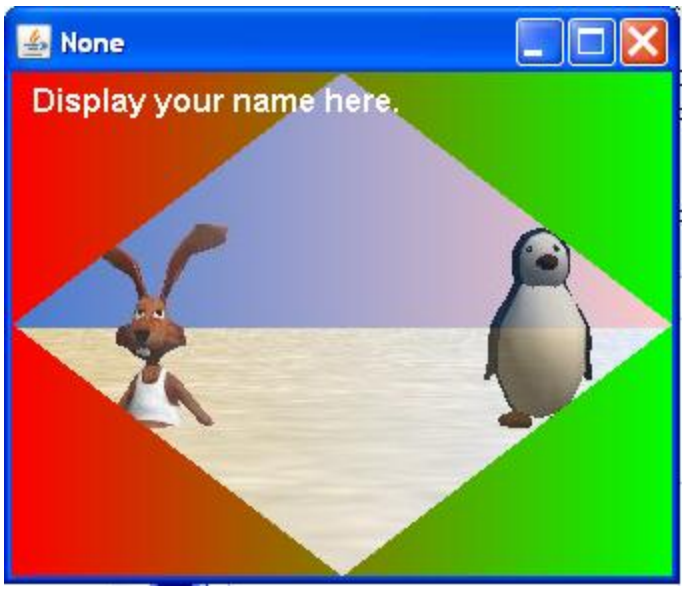
Go to answer 4

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.
- Figure 3 . Question 1 actual output.
- Figure 4 . Question 2 program code.
- Figure 5 . Question 2 possible output.
- Figure 6 . Question 2 actual output.
- Figure 7 . Question 3 program code.
- Figure 8 . Question 3 possible output.
- Figure 9 . Question 9.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 4

True.

Go back to Question 4

Answer 3

True.

Go back to Question 3

Answer 2

False. The output is shown in Figure 6 . The indentation in Figure 4 is not correct for the output shown in Figure 5 .

Figure 6 . Question 2 actual output.

```
['a', 'b', 'c']
6
['a', 'b', 'c', 6]
```

Go back to Question 2

Answer 1

False. The output is shown in Figure 3 . The code in the body of the while loop is never executed because the conditional clause returns false the first time it is tested.

Figure 3 . Question 1 actual output.

```
['a', 'b', 'c']
['a', 'b', 'c']
```

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1210r-Review
- File: Itse1359-1210r.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1220-Operators
This module presents and explains some of the many operators used in Python.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- Operators
    - Arithmetic operators
    - Relational operators
    - Logical operators
    - Bit shift operators
    - Miscellaneous operators
- Discussion and sample code
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Early modules in this collection included instruction on the following topics:

- numbers
- variables
- identifiers
- strings
- scripts
- lists

This instruction has prepared you to use Python as a very fancy desk calculator.

A more recent module provided instruction on the concepts of *repetition* and *decision logic* using the **while** loop as an example.

What you will learn

Prior to encountering the conditional clause in the **while** loop, all of the code involving operators was fairly self explanatory:

- addition
- subtraction
- multiplication
- division
- assignment

However, when examining the **while** loop, we were confronted with an operator that was less self explanatory -- the l *ess than or equal* operator. This suggests that this is the point in this collection where we need to describe all, or at least most of the operators used in Python. That is the purpose of this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Arithmetic operators.
- Figure 2 . Relational operators.
- Figure 3 . Logical operators.
- Figure 4 . Bit shift operators.
- Figure 5 . Miscellaneous operators.

Listings

- Listing 1 . Logical operators "and", "or", and "not".

Operators

The material on operators contained in this module was mainly extracted from 10.3.1. Mapping Operators to Functions in The Python Standard Library .

Arithmetic operators

As the name implies, the arithmetic operators are used for performing arithmetic. However, some of the operators are "overloaded" meaning that the behavior of the operator depends on the types of the operands. For example, as you learned in an earlier module, the (+) operator is used both for arithmetic addition and string concatenation.

Figure 1 shows all or at least most of the arithmetic operators.

Figure 1 . Arithmetic operators.

```
+    Addition          a + b
+    Positive          + a     (does
essentially nothing)
-    Subtraction       a - b
-    Negation          - a     (change sign of
operand)
*    Multiplication    a * b
**   Exponentiation    a ** b
/    Division          a / b   (floating point
division)
//   Division          a // b  (integer
division)
%    Modulo            a % b
```

With the exception of (//) and (%), the behavior of all of the operators in Figure 1 should be self-explanatory. The behavior of those two operators was explained in the earlier module titled *Itse1359-1020-Numbers* .

Relational operators

These are the operators that are typically used in the conditional clause of decision logic. Basically the operators test to determine if the operator describes the relationship between the left and right operands. If so, the expression returns true. Otherwise, it returns false.

Figure 2 shows all, or at least most of the relational operators.

Figure 2 . Relational operators.

```
<    a < b    (true if a is less than b)
<=   a <= b   (true if a is less than or equal
to b)
==   a == b   (true if a is equal to b)
!=   a != b   (true if a is not equal to b)
>=   a >= b   (true if a is greater than or
equal to b)
>    a > b    (true if a is greater than b)
```

Logical operators

The logical operators are shown in Figure 3 .

Figure 3 . Logical operators.

Figure 3 . Logical operators.

```
and   And           a and b   (true if both a
and b are true)
or    Inclusive Or  a or b    (true if either
a or b are true)
not   Negation      not a      (switch a from
true to false
                                or from false
to true)

&     Bitwise And           a & b
^     Bitwise Exclusive Or  a ^ b
|     Bitwise Inclusive Or  a | b
~     Bitwise Inversion     ~ a
```

The operators in Figure 3 are often used to embellish the relational operators from Figure 2 in decision logic.

The first operator *(**and**)* returns true if both operands are true.

The second operator *(**or**)* returns true if either operand is true.

The third operator *(**not**)* has only one operand and it is on the right side of the operator. It switches the operand from true to false or from false to true.

The script in Listing 1 shows examples of the use of these three operators in conjunction with the equality relational operator.

Listing 1 . Logical operators "and", "or", and "not".

```
a=5
print(a==5 and a==5)
print(a==5 and a==4)
print(a==5 and not a==5)
print(a==5 and not a==4)

print(a==5 or a==5)
print(a==5 or a==4)
print(a==4 or a==4)

#This script produces the following output:
True
False
False
True
True
True
False
```

The last four operators in Figure 3 are used to perform logical operations at the bit level. An explanation of these operators is beyond the scope of this module.

Bit shift operators

The operators in Figure 4 are used to shift bits to the right or left. As with the bitwise logical operators, a discussion of the behavior of these operators is beyond the scope of this module.

Figure 4 . Bit shift operators.

```
Left Shift    a << b
Right Shift   a >> b
```

Miscellaneous operators

Figure 5 shows a variety of miscellaneous operators. You are already familiar with the behavior of the concatenation operator. The behavior of the remaining operators will be explained when and if they are used in future modules.

Figure 5 . Miscellaneous operators.

Figure 5 . Miscellaneous operators.

```
Concatenation       seq1 + seq2

Containment Test    obj in seq
Identity            a is b
Identity            a is not b
Indexed Assignment  obj[k] = v
Indexed Deletion    del obj[k]
Indexing            obj[k]
Slice Assignment    seq[i:j] = values
Slice Deletion      del seq[i:j]
Slicing seq[i:j]
String Formatting   s % obj
Truth Test          obj truth(obj)
```

Discussion and sample code

Other than the code shown in Listing 1 , I won't present code to illustrate the use of these operators in this module. Instead, I will use the operators in code in future modules and discuss them, as appropriate when they are used.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1220-Operators
- File: Itse1359-1220.htm

- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1220r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1220-Operators.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1220-Operators.*

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
count = 0
while count <= 5 and 3*count//2 <=5:
  count = count + 1
  print(count)

print(count)
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
1
2
3
4
4
```

Go to answer 1

Question 2

True or False? The code in Figure 3 produces the output shown in Figure 4 .

Figure 3 . Question 2 program code.

```
count = 0
while count < 5 or 2*count+1 < 15:
  count = count + 1
  print(count)

print(count)
```

Figure 4 . Question 2 possible output.

```
1
2
3
4
5
6
7
7
```

Go to answer 2

Question 3

True or False? The code in Figure 5 produces the output shown in Figure 6 .

Figure 5 . Question 3 program code.

Figure 5 . Question 3 program code.

```
count = 0
while not(count >= 5 and 2*count+1 >= 15):
  count = count + 1
  print(count)

print(count)
```

Figure 6 . Question 3 possible output.

```
1
2
3
4
5
6
7
8
8
```

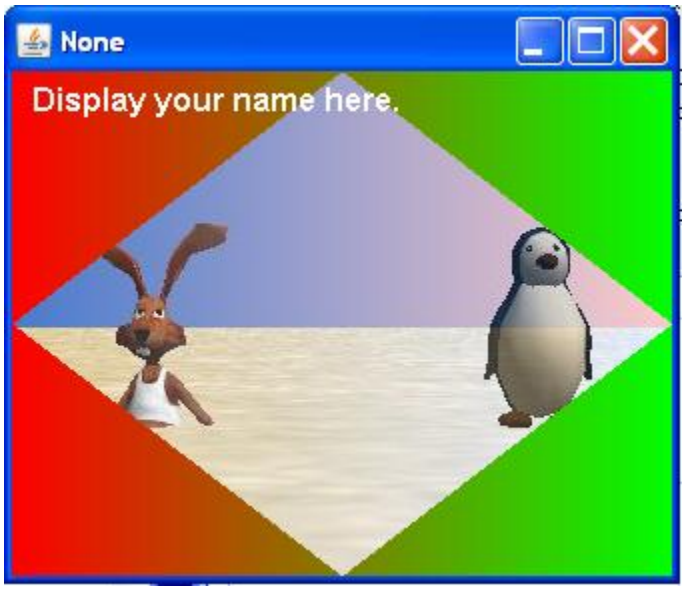
Go to answer 3

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.
- Figure 3 . Question 2 program code.
- Figure 4 . Question 2 possible output.
- Figure 5 . Question 3 program code.
- Figure 6 . Question 3 possible output.
- Figure 7 . Question 3 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 3

False. The actual output is shown in Figure 7 .

Figure 7 . Question 3 actual output.

Figure 7 . Question 3 actual output.

```
1
2
3
4
5
6
7
7
```

Go back to Question 3

Answer 2

True.

Go back to Question 2

Answer 1

True.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1220r-Review
- File: Itse1359-1220r.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1230r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1230-The if Statement

Table of contents

- Preface
- Questions
    - 1 , 2 , 3
- Image index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1230-The if Statement* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Image 1 produces the output shown in Image 2 .

Image 1 . Question 1 program code.

```
dogs = ["Affenpinscher","Afgan Hound","Akita"]

if len(dogs) != 2:
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])
print("Length of dogs list = " +
str(len(dogs)))

if not(len(dogs) == 3):
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])
print(dogs)
```

Image 2 . Question 1 possible output.

```
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
['Affenpinscher', 'Afgan Hound', 'Akita']
```

Go to answer 1

Question 2

True or False? The code in Image 3 produces the output shown in Image 4 .

Image 3 . Question 2 program code.

Image 3 . Question 2 program code.

```
weather = ["sunshine","rain"]

weatherToday = weather[1]

if weatherToday == "rain":
  print("It's raining, visit the museum.")
else:
  print("Sunshine, go to the beach.");

print("Vacation is over, go home.")
```

Image 4 . Question 2 possible output.

```
Sunshine, go to the beach.
Vacation is over, go home.
```

Go to answer 2

Question 3

True or False? The code in Image 6 produces the output shown in Image 7 .

Image 6 . Question 3 program code.

```
weather = ["sunshine","rain","snow"]

weatherToday = weather[2]

if weatherToday == "rain":
  print("It's raining, visit the museum.")
elif weatherToday == "sunshine":
  print("Sunshine, go to the beach.")
else:
  print("It's snowing, go skiing.")

print("Vacation is over, go home.")
```

Image 7 . Question 3 possible output.

```
It's snowing, go skiing.
Vacation is over, go home.
```

Go to answer 3

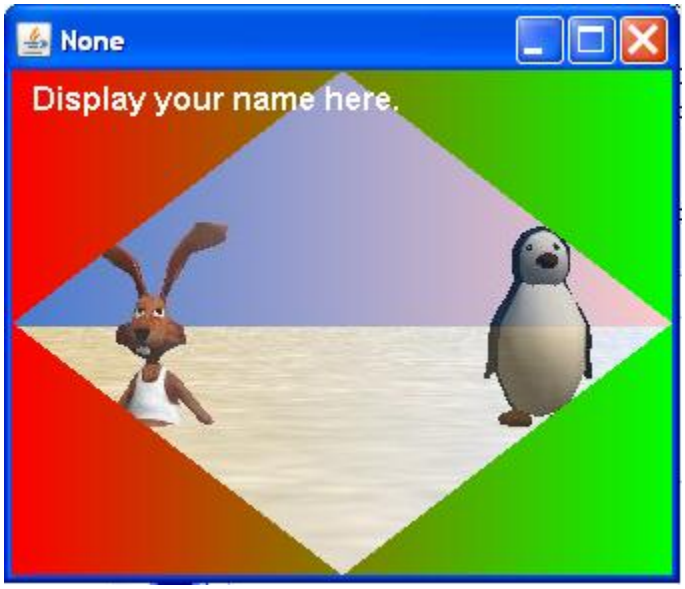
Image index

- Image 1 . Question 1 program code.

- Image 2 . Question 1 possible output.
- Image 3 . Question 2 program code.
- Image 4 . Question 2 possible output.
- Image 5 . Question 2 actual output.
- Image 6 . Question 3 program code.
- Image 7 . Question 3 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 3

True.

Go back to Question 3

Answer 2

False. The actual output is shown in Image 5 .

Image 5 . Question 2 actual output.

Image 5 . Question 2 actual output.

```
It's raining, visit the museum.
Vacation is over, go home.
```

Go back to Question 2

Answer 1

True.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1230r-Review
- File: Itse1359-1230r.htm
- Published: 10/21/14
- Revised: 12/28/14

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1240r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1240-The for Loop.

Table of contents

- Preface
- Questions
    - 1 , 2
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1240-The for Loop* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
dogs = ["Affenpinscher","Afgan Hound","Akita"]

for breed in dogs:
  print(breed)
print("Length of dogs list = " +
str(len(dogs)))

for letter in dogs[2]:
  print(letter)
print(dogs[2])
```

Figure 2 . Question 1 possible output.

Figure 2 . Question 1 possible output.

```
Akita
Afgan Hound
Affenpinscher
Length of dogs list = 3
A
k
i
t
a
Akita
```

Go to answer 1

Question 2

True or False? The code in Figure 4 produces the output shown in Figure 5 .

Figure 4 . Question 2 program code.

Figure 4 . Question 2 program code.

```
breed = "Dachshund"
for index in range(2,len(breed),3):
  print("Letter at index " + str(index) + " is
" + breed[index])
print(breed)

sum = 0
for cnt in range(5):
  sum += 2*cnt
  print("cnt = " + str(cnt) + ", sum = " +
str(sum))
```

Figure 5 . Question 2 possible output.

```
Letter at index 2 is c
Letter at index 5 is h
Letter at index 8 is d
Dachshund
cnt = 0, sum = 0
cnt = 1, sum = 2
cnt = 2, sum = 6
cnt = 3, sum = 12
cnt = 4, sum = 20
```

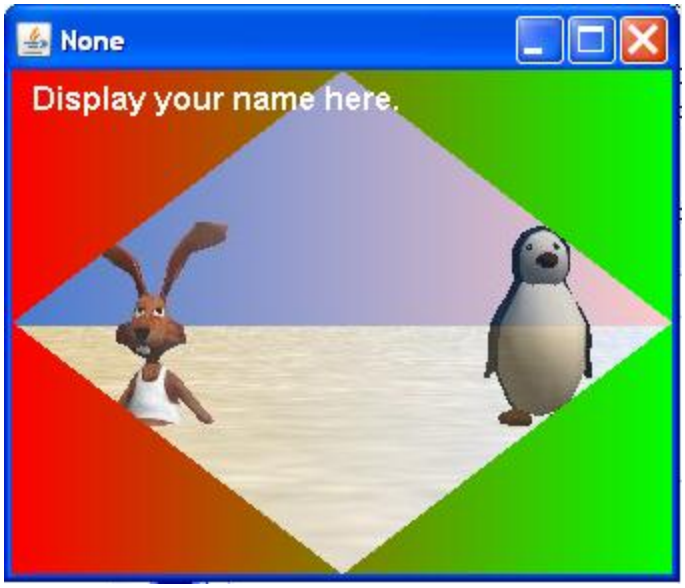
Go to answer 2

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.
- Figure 3 . Question 1 actual output.
- Figure 4 . Question 2 program code.
- Figure 5 . Question 2 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 2

True.

Go back to Question 2

Answer 1

False. The actual output is shown in Figure 3 .

Figure 3 . Question 1 actual output.

Figure 3 . Question 1 actual output.

```
Affenpinscher
Afgan Hound
Akita
Length of dogs list = 3
A
k
i
t
a
Akita
```

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1240r-Review
- File: Itse1359-1240r.htm
- Published: 10/21/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1250-Nested Loops
This module explains nested loops in Python.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- General background information
- Discussion and sample code
- Visualizing the program code
- Run the program
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You have learned how to construct **while** loops, **for** loops, and **if** statements in recent modules on *control flow* in Python.

What you will learn

In this module, you will learn that loops can be nested and how to nest them.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Syntax for nested while loops.
- Figure 2 . Syntax for nested for loops
- Figure 3 . Output from the code in Listing 1.

Listings

- Listing 1 . An example of nested while and for loops.

General background information

Python allows you to nest **while** loops and **for** loops inside of **while** loops and **for** loops to a greater depth than you are likely to need.

Figure 1 shows the general syntax for nested **while** loops.

Figure 1 . Syntax for nested while loops.

```
while expression:
  while expression:
    while expression:
      statement(s)
    statement(s)
  statement(s)
```

Figure 2 shows the general syntax of nested **for** loops.

Figure 2 . Syntax for nested for loops

```
for var in sequence:
   for var in sequence:
      statements(s)
   statements(s)
```

The syntax can become more complicated than that shown in either Figure 1 or Figure 2 . This is particularly true when you mix while loops and for loops in a nested construct.

Discussion and sample code

Listing 1 shows a program with a **while** loop nested inside of a **for** loop which in turn is nested inside of another **while** loop.

Listing 1 . An example of nested while and for loops.

```
# Illustrates nested loops--------------------
-----------------------------

leftDigit = 0
rightDigit = 0

while leftDigit < 3:

  for middleDigit in range(3):
    rightDigit = 0
    print("") #blank line

    while rightDigit < 3:
      print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
      rightDigit += 1
    #end inner while loop

  #end for loop

  leftDigit += 1
#end outer while loop
```

Figure 3 shows the output from the code in Listing 1 . Note that the blank lines were inserted to make the material in Figure 3 easier to read.

Figure 3 . Output from the code in Listing 1.

```
0-0-0
0-0-1
0-0-2

0-1-0
0-1-1
0-1-2

0-2-0
0-2-1
0-2-2

1-0-0
1-0-1
1-0-2

1-1-0
1-1-1
1-1-2

1-2-0
1-2-1
1-2-2

2-0-0
```

Figure 3 . Output from the code in Listing 1.

```
2-0-1
2-0-2

2-1-0
2-1-1
2-1-2

2-2-0
2-2-1
2-2-2
```

This program simulates a scaled-down version of the odometer in your car. This odometer, however, only has three digits and each digit is limited to the values of 0, 1, and 2. In other words, Each digit of the odometer rolls over to 0 when its value exceeds 2.

The program begins by creating two working variables named **leftDigit** and **rightDigit** and initializing their values to zero. These are essentially counter variables for the outermost **while** loop and the innermost **while** loop in Listing 1 . They are also the left-most and right-most digits shown in the columns of digits in Figure 3 .

It isn't necessary to create a separate working variable for the middle digit. That variable, named **middleDigit** is created as part of the **for** loop statement in Listing 1 .

If you compare the code in Listing 1 with the output in Figure 3 , you will see that the **for** loop goes through its full range of iterations (3) for each iteration of the outermost **while** loop.

You will also see that the innermost **while** loop goes through its full range of iterations (3) for every iteration of the **for** loop. As a result, the innermost **while** loop goes through nine iterations for every iteration of the outermost **while** loop.

When you think about the behavior of nested loops, think of an odometer. That may help you to remember how they behave.

Visualizing the program code

I recommend that you create a visualization for the code in Listing 1 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of nested loops.

Run the program

I also encourage you to copy the code from Listing 1. Execute the code and confirm that you get the same results as those shown in Figure 3. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do. For example, rewrite the program using all **while** loops, all **for** loops, or some other combination of **while** loops and **for** loops.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1250-Nested Loops
- File: Itse1359-1250.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1250r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1250-Nested Loops

Table of contents

- Preface
- Questions
    - 1
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1250-Nested Loops* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 .

Figure 1 . Question 1 program code.

```
leftDigit = 0
rightDigit = 0

while leftDigit < 3:

  for middleDigit in range(3,0,-1):
    rightDigit = 0
    print("") #blank line

    while rightDigit < 3:
      print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
      rightDigit += 1
    #end inner while loop

  #end for loop

  leftDigit += 1
#end outer while loop
```

Figure 2 . Question 1 possible output.

```
0-0-0
0-0-1
0-0-2

0-1-0
0-1-1
0-1-2

0-2-0
0-2-1
0-2-2

1-0-0
1-0-1
1-0-2

1-1-0
1-1-1
1-1-2

1-2-0
1-2-1
1-2-2

2-0-0
2-0-1
2-0-2

2-1-0
2-1-1
2-1-2

2-2-0
```

Figure 2 . Question 1 possible output.

```
2-2-1
2-2-2
```

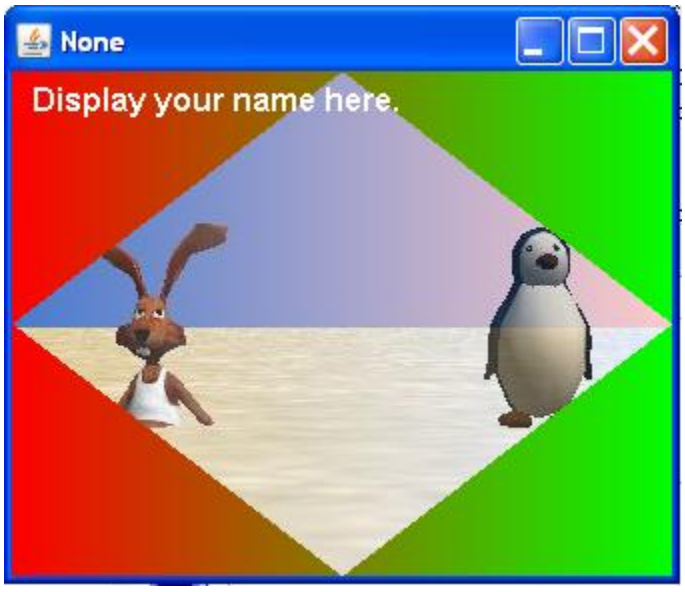
Go to answer 1

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.
- Figure 3 . Question 1 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 1

False. The actual output is shown in Figure 3 .

Figure 3 . Question 1 actual output.

```
0-3-0
0-3-1
0-3-2

0-2-0
0-2-1
0-2-2
```

Figure 3 . Question 1 actual output.

```
0-1-0
0-1-1
0-1-2

1-3-0
1-3-1
1-3-2

1-2-0
1-2-1
1-2-2

1-1-0
1-1-1
1-1-2

2-3-0
2-3-1
2-3-2

2-2-0
2-2-1
2-2-2

2-1-0
2-1-1
2-1-2
```

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1250r-Review
- File: Itse1359-1250r-Review.htm
- Published: 10/26/14
- Revised: 02/25/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1260-Loop Modifiers
This module explains the Python loop modifiers: else, continue, break,and pass.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- Discussion and sample code
    - The else clause
    - Visualize loops with else clauses
    - The continue statement
    - Visualize the behavior of the continue statement
    - The break statement
    - Visualize the behavior of the break statement
    - The pass statement
- Run the program
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You have learned how to construct **while** loops, **for** loops, and **if** statements in recent modules on *control flow* in Python. In addition, you recently learned how to create *nested loop structures* .

What you will learn

In this module, you will learn how to add an **else** clause to **while** loops and **for** loops. In addition, once you start using nested loops, you need to know how to use the loop modifiers: **continue** and **break** . Those loop modifiers will be explained in this module.

Finally, this module will make a brief mention of the **pass** statement, which does nothing. If you need to write code that does nothing, the **pass** statement is for you.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Output from the code in Listing 1.
- Figure 2 . Output from the code in Listing 2.
- Figure 3 . Output from the code in Listing 3.

Listings

- Listing 1 . Loops with else clauses.
- Listing 2 . Nested loops with a continue statement.
- Listing 3 . Nested loops with a break statement.

Discussion and sample code

The else clause

You learned about the **else** clause on an **if** statement in an earlier module titled *Itse1359-1230-The if Statement* .

Loop statements can also have an **else** clause. The **else** clause on a **for** loop is executed when the loop terminates through exhaustion of the elements in the list. The **else** clause on a **while** loop is executed when the loop terminates because the condition becomes **False** . The **else** clause is not executed when **the loop is terminated by a** break statement.

I will illustrate the **else** clause using a program that is similar to the odometer program that you learned about in the earlier module titled Itse1359-1250-Nested Loops . The program is shown in Listing 1 .

Listing 1 . Loops with else clauses.

```
# Illustrates nested loops with else clauses
#-----------------------------------------------
------------------------------

leftDigit = 0
```

Listing 1 . Loops with else clauses.

```
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
  leftDigit += 1

  #Begin for loop
  for middleDigit in range(1,3):
    rightDigit = 0

    #Begin innermost while loop
    while rightDigit < 3:
      rightDigit += 1
      print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
    else:
      print("In else clause on inner while
loop")
    #end inner while loop with else clause


  else:
    print("In else clause on for loop")
  #end for loop with else clause

else:
  print("In else clause on outer while loop")
#end outer while loop with else clause

print("Goodbye")
```

The output from the code in Listing 1 is shown in Figure 1 .

Figure 1 . Output from the code in Listing 1.

```
1-1-1
1-1-2
1-1-3
In else clause on inner while loop
1-2-1
1-2-2
1-2-3
In else clause on inner while loop
In else clause on for loop
2-1-1
2-1-2
2-1-3
In else clause on inner while loop
2-2-1
2-2-2
2-2-3
In else clause on inner while loop
In else clause on for loop
In else clause on outer while loop
Goodbye
```

You should already be familiar with most of the code in Listing 1 . Some of the statements have been rearranged relative to the odometer code in the earlier module and some of the limits in the conditions have been reduced. However, that was done simply to shorten and to improve the clarity of the printed output and is of no technical consequence.

The only thing that is new in Listing 1 is the addition of an **else** clause to each of the loops. Note that the indentation level of the word **else** in each case is the same as the indentation level for the corresponding words **while** and **for** . For example, the last **else** clause in Listing 1 belongs to the

outermost **while** loop that begins with the word **while** near the top of Listing 1 .

In this program, each of the **else** clauses simply prints a message indicating that it is being executed. Obviously, more substantive code could be placed in the **else** clause in a more significant program.

The innermost **while** loop in this program is executed twice for each iteration of its enclosing **for** loop. Similarly, the **for** loop is executed twice for each iteration of its enclosing **while** loop. The outermost **while** loop is executed once, the **for** loop is executed twice, and the innermost **while** loop is executed **four** times.

The **else** clause for a loop is executed each time the loop terminates *(with the exception of break as mentioned earlier)* . As you can see in Figure 1 , the **else** clause on the innermost **while** loop was executed four times, the **else** clause on the **for** loop was executed two times, and the **else** clause on the outermost **while** loop was executed once.

Visualize loops with else clauses

I recommend that you create a visualization for the code in Listing 1 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of loops with else clauses.

The continue statement

If you execute a **continue** statement inside a **while** loop or a **for** loop, that will cause the current iteration of the loop to terminate and the next iteration to begin.

> (Be careful that the code that updates the condition in a while loop is not bypassed by a continue statement. If you do, you will end up with an infinite loop.)

The use of a **continue** statement is illustrated in Listing 2 , which shows an **if** statement *(with a **continue** statement in the body of the **if** statement)* inserted into the innermost **while** loop of the program from Listing 1 .

Listing 2 . Nested loops with a continue statement.

```
# Illustrates nested loops with else and
continue
# ----------------------------------------
-----------------------------

leftDigit = 0
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
  leftDigit += 1

  #Begin for loop
  for middleDigit in range(1,3):
    rightDigit = 0

    #Begin innermost while loop
    while rightDigit < 3:
```

Listing 2 . Nested loops with a continue statement.

```
      rightDigit += 1
      if rightDigit == 2:
        print("continue")
        continue
      print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
    else:
      print("In else clause on inner while
loop")
    #end inner while loop with else clause


  else:
    print("In else clause on for loop")
  #end for loop with else clause

else:
  print("In else clause on outer while loop")
#end outer while loop with else clause

print("Goodbye")
```

Figure 2 shows the output produced by the code in Listing 2 . You should compare this output with the output shown in Figure 1 .

Figure 2 . Output from the code in Listing 2.

Figure 2 . Output from the code in Listing 2.

```
1-1-1
continue
1-1-3
In else clause on inner while loop
1-2-1
continue
1-2-3
In else clause on inner while loop
In else clause on for loop
2-1-1
continue
2-1-3
In else clause on inner while loop
2-2-1
continue
2-2-3
In else clause on inner while loop
In else clause on for loop
In else clause on outer while loop
Goodbye
```

Once during each iteration of the **while** loop in Listing 2 , a test is made to determine if the condition variable, **rightDigit** , is equal to 2. If so, the word *continue* is printed and a **continue** statement is executed.

The execution of the **continue** statement causes the current iteration of the loop to terminate and the next iteration to begin. This, in turn causes the **print** statement following the **if** statement to be skipped and the word *continue* to be printed in its place. You can easily see where this happens by comparing the output in Figure 2 with the output in Figure 1 .

Visualize the behavior of the continue statement

I also recommend that you create a visualization for the code in Listing 2 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of the **continue** statement.

The break statement

The **break** statement is much more drastic than the **continue** statement. Whereas a **continue** statement causes the current iteration of the loop to terminate, a **break** statement causes the entire loop to terminate.

This is illustrated in Listing 3 , which shows a **break** statement inserted in place of the **continue** statement from Listing 2 . The **print** statement was also modified, causing it to print *break* in place of *continue* .

Listing 3 . Nested loops with a break statement.

```
# Illustrates nested loops with else and break
#-------------------------------------------
-----------------------------

leftDigit = 0
rightDigit = 0

#Begin outer while loop
while leftDigit < 2:
```

Listing 3 . Nested loops with a break statement.

```
  leftDigit += 1

  #Begin for loop
  for middleDigit in range(1,3):
    rightDigit = 0

    #Begin innermost while loop
    while rightDigit < 3:
      rightDigit += 1
      if rightDigit == 2:
        print("break")
        break
      print(str(leftDigit) + "-" +
str(middleDigit) + "-" + str(rightDigit))
    else:
      print("In else clause on inner while
loop")
    #end inner while loop with else clause


  else:
    print("In else clause on for loop")
  #end for loop with else clause

else:
  print("In else clause on outer while loop")
#end outer while loop with else clause

print("Goodbye")
```

The output from the code in Listing 3 is shown in Figure 3 . You should compare this with the output shown in Figure 2 .

Figure 3 . Output from the code in Listing 3.

```
1-1-1
break
1-2-1
break
In else clause on for loop
2-1-1
break
2-2-1
break
In else clause on for loop
In else clause on outer while loop
Goodbye
```

The output for the **continue** statement in Figure 2 shows that only one iteration of the innermost **while** loop was impacted by the **if** statement and the **continue** statement in its body.

The output shown in Figure 3 shows that once the **break** statement was executed, no further iterations of the innermost **while** loop were executed. In other words, execution of the **break** statement terminated the loop in its entirety.

Figure 3 also shows that termination caused by a **break** statement also prevented the execution of the code in the **else** clause for the innermost **while** loop as indicated earlier ..

The innermost **while** loop is still executed twice for each iteration of its enclosing **for** loop. However, the **while** loop terminates part of the way through the second iteration and the printed output that would be produced by the second and third iterations *(see Listing 3)* is missing from the output shown in Figure 3 .

Visualize the behavior of the break statement

I also recommend that you create a visualization for the code in Listing 3 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the behavior of the **break** statement.

The pass statement

I promised that this module will make brief mention of the **pass** statement, which does nothing. This is that brief mention. If you need to write code that does nothing, the **pass** statement will do that for you.

Run the program

I encourage you to copy the code from Listing 1 , Listing 2 , and Listing 3 . Execute the code and confirm that you get the same results as those shown in Figure 1 , Figure 2 , and Figure 3 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

I also recommend that you create and experiment with visualizations for the code in Listing 1 , Listing 2 , and Listing 3 .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1260-Loop Modifiers
- File: Itse1359-1260.htm
- Published: 10/26/14

- Revised: 09/05/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1280-Function Arguments
This module explains how to tap into some of the more-powerful capabilities of Python functions using required arguments, default arguments, keyword arguments, and variable-length arguments.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- General background information
- Discussion and sample code
    - Passing by value or reference
        - The function definition
        - A non-technical explanation
        - Visualization of the code in Listing 1
        - A modified approach to the same objective
    - Required arguments
    - Default arguments
    - Keyword arguments
    - Variable-length arguments
        - Visualization of variable-length arguments
- Run the programs
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

You learned the basics of defining and calling Python functions in the earlier module titled Itse1359-1270-Functions .

What you will learn

In this module, you will learn how to tap into some of the more-powerful capabilities of functions involving optional ways to use arguments. This will include an explanation of

- required arguments,
- default arguments,
- keyword arguments, and
- variable-length arguments.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Output from the code in Listing 1.
- Figure 2 . Visualization of the code in Listing 1.
- Figure 3 . Output from the code in Listing 2.
- Figure 4 . Output produced by the code in Listing 3.
- Figure 5 . Output produced by the code in Listing 4.
- Figure 6 . Output produced by the code in Listing 5.
- Figure 7 . Visualization of variable-length arguments.

Listings

- Listing 1 . A list-modifier function.
- Listing 2 . Another list-modifier function.
- Listing 3 . Illustration of default arguments.
- Listing 4 . Illustration of keyword arguments.
- Listing 5 . Illustration of variable-length arguments.

General background information

The overall topic of function arguments in Python is broad and relatively complex. A complete treatment of the topic is beyond the scope of this collection of modules. Instead, I will provide some examples of what is possible. For a more compete treatment, I will refer you to The Python Tutorial -- More on Defining Functions and tutorialspoint -- Python Functions .

One of the fundamental concepts in computer programming is whether function parameters are passed *by value* or *by reference* . If a parameter is passed by value, the function receives a copy of the thing referred to by the parameter. Modifying that thing in the function will not modify the original.

If a parameter is passed by reference, the incoming parameter can be used by code in the function to modify the original. Some programming languages support both types of parameter passing.

According to tutorialspoint -- Python Functions , all parameters in Python are *passed by reference* . On the other hand, according to The Python Tutorial -- Defining Functions ,

> "thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object)."

In this case, I come down on the side of *call by value* . I think it is more correct to say that parameters are passed by value and as a result, the function receives copies of references to objects. Even though I can use a copy of an object's reference to modify the object, I cannot use a copy of that reference to cause the original reference to point to a different object. I can modify the object to which the reference points, but I cannot modify the reference itself.

I will illustrate what I mean by this in conjunction with the program in Listing 1 .

Discussion and sample code

This module will examine the following kinds of arguments:

- Required arguments
- Default arguments
- Keyword arguments
- Variable-length arguments

Passing by value or reference

The program shown in Listing 1 illustrates the significance of passing parameters by *value* or by *reference* when those parameters are references to objects. Listing 1 also illustrates the use of two *required arguments* .

Listing 1 . A list-modifier function.

Listing 1 . A list-modifier function.

```
# Illustrates pass by value or reference
#-----------------------------------------
------

def listModifier(listA,listB):
  """Illustrates pass by value or reference"""
  print("In listModifier")

  print("Use incoming parameter to append to
listA")
  listA.append(3.14159)
  print("New listA = " + str(listA))

  print("Assign a new list to listB")
  listB = ["A","new","list"]
  print("New listB = " + str(listB))

  return
#End function definition

#Call the function
print("Create two lists")
aList = ["ab","cd","ef"]
bList = ["The","old","list"]

print("aList = " + str(aList))
print("bList = " + str(bList))

print("Call listModifier")
listModifier(aList,bList)
print("Back from listModifier")
print("aList = " + str(aList))
print("bList = " + str(bList))
```

Listing 1 . A list-modifier function.

Figure 1 shows the output produced by the code in Listing 1 .

Figure 1 . Output from the code in Listing 1.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['The', 'old', 'list']
```

The function definition

The code in Listing 1 defines a function named **listModifier** . This function receives two incoming parameters. Each parameter points to a different list. I contend that the incoming parameters are actually copies of the variables that were passed as parameters.

The code in the function uses one of the incoming parameters to access the list and to append a value of 3.14159 onto the end of the list. The second

line of text from the bottom of Figure 1 confirms that this operation was successful.

Then the code in the function creates a new list and attempts to use the second incoming parameter to replace the original list pointed to by that parameter with the new list. The last line of text in Figure 1 confirms that this was **not successful** . The copy of the incoming parameter points to the new list but this does not cause the original reference to point to the new list.

A non-technical explanation

This same situation occurs in the Java courses that I teach. I often attempt to explain it this way. Assume that I write my home address on a piece of paper and make a copy of that piece of paper. Then I lose the copy. If a burglar finds the copy, he has a pointer to my house. He could use that pointer in an attempt to break into my house. However, even if he were to scratch out the address on the copy and write a new address on the copy, that would not change the address written on the original. The original would still point to the same old address.

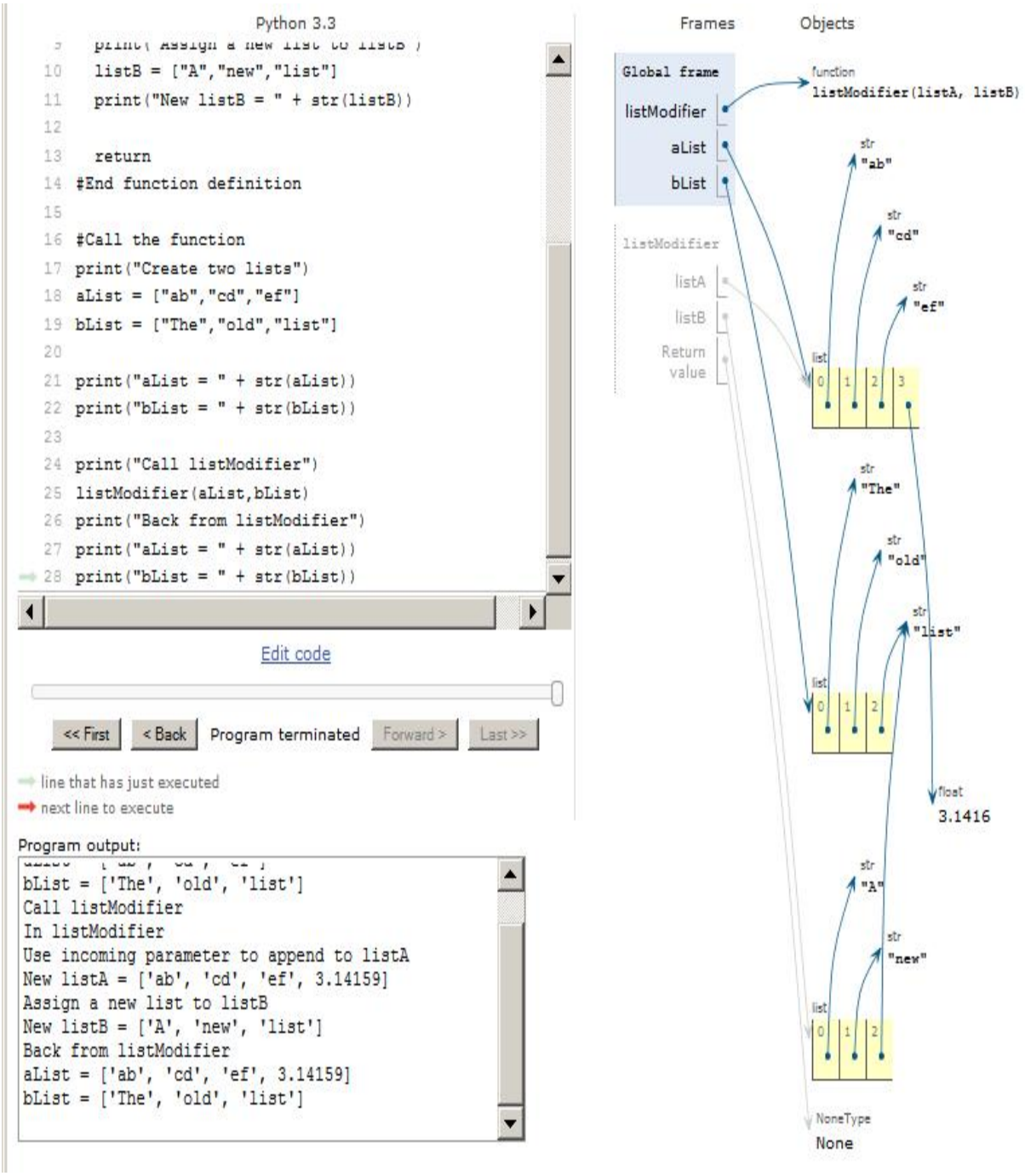
This is essentially what happens in Listing 1 when the code creates a new list and assigns its value to the copy of the original reference. The copy points to the new list inside the function but the original reference outside the function continues to point to the original list.

In effect, the copies that are received as parameters by a function become local variables within the function. Those variables, and in this case the new list that is pointed to by one of those variables, all cease to exist when the function terminates unless returned by the function.

Visualization of the code in Listing 1

Figure 2 shows a visualization of the code in Listing 1 after all of the code has been executed.

Figure 2. Visualization of the code in Listing 1.



I recommend that you create a visualization for the code in Listing 1 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the

diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of *"passing by value."*

A modified approach to the same objective

Listing 2 shows a modification to the code from Listing 1 that is designed to accomplish what Listing 1 attempted to do but was unable to do.

Listing 2 . Another list-modifier function.

```
# Illustrates pass by value or reference
#----------------------------------------
------

def listModifier(listA,listB):
  """Illustrates pass by value or reference"""
  print("In listModifier")

  print("Use incoming parameter to append to
listA")
  listA.append(3.14159)
  print("New listA = " + str(listA))

  print("Assign a new list to listB")
  listB = ["A","new","list"]
  print("New listB = " + str(listB))

  return listB
#End function definition
```

Listing 2 . Another list-modifier function.

```
#Call the function
print("Create two lists")
aList = ["ab","cd","ef"]
bList = ["The","old","list"]

print("aList = " + str(aList))
print("bList = " + str(bList))

print("Call listModifier")
bList = listModifier(aList,bList)
print("Back from listModifier")
print("aList = " + str(aList))
print("bList = " + str(bList))
```

Figure 3 shows the output from the code in Listing 2 .

Figure 3 . Output from the code in Listing 2.

Figure 3 . Output from the code in Listing 2.

```
Create two lists
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
Call listModifier
In listModifier
Use incoming parameter to append to listA
New listA = ['ab', 'cd', 'ef', 3.14159]
Assign a new list to listB
New listB = ['A', 'new', 'list']
Back from listModifier
aList = ['ab', 'cd', 'ef', 3.14159]
bList = ['A', 'new', 'list']
```

Listing 2 has two major modifications relative to Listing 1 . First, the function returns a copy of the reference to the new list instead of returning nothing. Second, the return value is assigned to the variable named **bList** . This does cause the variable named **bList** to point to a different list object. This is confirmed by the last line of text in Figure 3 .

Conclusion -- function parameters in Python are passed by value, but the copies that are passed can often be used to access and modify the objects pointed to by the original reference variables of which they are copies.

In this program, the function's second argument is of no consequence. The function and the program would behave the same if the second argument were eliminated entirely. I left it in for illustration purposes only.

Hopefully you can study these two programs along with the program output and the above discussion to understand the basic behavior of Python function arguments.

Required arguments

Note that there is a subtle technical difference between arguments and parameters. However, I often tend to use the two terms interchangeably so you will simply need to grit your teeth and bear with me if you see me using one or the other term incorrectly.

When a function defines one or more arguments using the syntax shown in Listing 1 and Listing 2 , the code that calls the function must pass one parameter for each argument and must pass them in the correct order. That is why we call them *required arguments* . There isn't a lot more that needs to be said about required arguments.

Default arguments

A function can define one or more default arguments in addition to required arguments as shown in Listing 3 . Default values are defined for the default arguments when the function is defined using the syntax shown in Listing 3 .

The calling program can ignore some or all of the default arguments when calling the function. By this I mean that the calling program can simply not pass parameters for some or all of the default arguments. However, when default arguments are ignored, they must be ignored from right to left in the argument list. In other words, the calling program cannot ignore a default argument in the middle of a group of default arguments.

When the calling program ignores default arguments, the defined default values for those arguments are used by the code in the body of the function.

The program shown in Listing 3 illustrates a function with one required argument (**listA**) and three default arguments (**listB** , **listC** , and **listD**).

Listing 3 . Illustration of default arguments.

```
# Illustrates default arguments
#-----------------------------------------
-------------

def listModifier(listA,listB=["B"],listC=
["C"],listD=["D"]):
  """Illustrates default arguments"""
  print("In listModifier")

  listA.append(1.00001)
  print("listA = " + str(listA))
  listB.append(2.00002)
  print("listB = " + str(listB))
  listC.append(3.00003)
  print("listC = " + str(listC))
  listD.append(4.00004)
  print("listD = " + str(listD))

  return
#End function definition

aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,bList)
```

Listing 3 . Illustration of default arguments.

```
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))
```

Figure 4 shows the output produced by the code in Listing 3 .

Figure 4 . Output produced by the code in Listing 3.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
listA = ['ab', 'cd', 'ef', 1.00001]
listB = ['The', 'old', 'list', 2.00002]
listC = ['C', 3.00003]
listD = ['D', 4.00004]
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house']
dList = ['is falling down']
```

The first thing that you should pay attention to is the syntax for defining default arguments in the function named **listModifier** . *(You will see later that this **same syntax is used for defining keyword arguments** .)*

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of Listing 3 . The function call passes only two parameters to the function: **aList** and **bList** .

The function treats the first parameter as satisfying the required argument named **listA** . The function treats the second parameter as satisfying the default argument named **ListB** , replacing the default value with the parameter that is actually received. Not receiving any more incoming parameters, the function processes the default values for **listC** and **listD** .

Hopefully this explanation along with the code in Listing 3 and the output in Figure 4 will tell you what you need to know about using default function arguments.

Keyword arguments

When a function having required arguments is called, the calling program must pass a parameter for each required argument and must pass those parameters in the correct left-to-right order relative to the argument list.

When a function having default arguments is called, the calling program can omit passing parameters to the arguments on the right end of the argument list. However, the parameters that are passed to default arguments must be passed in the correct left-to-right order relative to the argument list.

You are probably seeing the trend here. When a function having keyword arguments is called, the calling program can pass parameters for all, some, or none of the keyword arguments. Furthermore, the parameters that are passed to keyword arguments can be passed in any order. Positional matching is no longer required.

A function can define required arguments, default arguments, keyword arguments, and variable-length arguments *(to be discussed later)* in a variety of combinations. In this module, I will combine required arguments with each of the other types but won't provide examples of combining all four types in the same function.

A function can define one or more keyword arguments in addition to required arguments as shown in Listing 4 . Default values are defined for the keyword arguments when the function is defined using the syntax shown in Listing 4 . *(I told you earlier that the syntax for defining default arguments is the same as the syntax for defining keyword arguments.)*

The calling program can ignore none, some, or all of the keyword arguments when calling the function. By this I mean that the calling program can simply not pass parameters for keyword arguments. Parameters that are passed for keyword arguments can be passed in any order. In other words, unlike default arguments, the calling program can ignore a keyword argument in the middle of a group of keyword arguments.

As with default arguments, when the calling program ignores keyword arguments, the defined default values for those arguments are used by the code in the body of the function.

The program shown in Listing 4 illustrates a function with one required argument (**listA**) and three keyword arguments (**listB** , **listC** , and **listD**).

Listing 4 . Illustration of keyword arguments.

```
# Illustrates keyword arguments
#-----------------------------------------
------
```

Listing 4 . Illustration of keyword arguments.

```
def listModifier(listA,listB=["B"],listC=
["C"],listD=["D"]):
  """Illustrates keyword arguments"""
  print("In listModifier")

  listA.append(1.00001)
  print("listA = " + str(listA))
  listB.append(2.00002)
  print("listB = " + str(listB))
  listC.append(3.00003)
  print("listC = " + str(listC))
  listD.append(4.00004)
  print("listD = " + str(listD))

  return
#End function definition

aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,listD=dList,listB=bList)
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
```

Listing 4 . Illustration of keyword arguments.

```
print("cList = " + str(cList))
print("dList = " + str(dList))
```

The code in Listing 4 produces the output shown in Figure 5 .

Figure 5 . Output produced by the code in Listing 4.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
listA = ['ab', 'cd', 'ef', 1.00001]
listB = ['The', 'old', 'list', 2.00002]
listC = ['C', 3.00003]
listD = ['is falling down', 4.00004]
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house']
dList = ['is falling down', 4.00004]
```

As before, the first thing that you should pay attention to is the syntax for defining keyword arguments in the function named **listModifier** in Listing 4 . If you compare it with the argument list for the function with the same

name in Listing 3 , you will see that there are **exactly the same** . Therefore, the manner in which the arguments are defined does not distinguish default arguments from keyword arguments. The distinguishing factor is the syntax with which the function is called.

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of Listing 4 . The function call passes three parameters: **aList** , **dList** , and **bList** .

The reference variable named **aList** is passed as the first parameter to satisfy the required argument named **listA** . Note that it is passed using the same syntax as in Listing 1 , Listing 2 , and Listing 3 . Because it is a required argument, it must be passed in the correct order in a positional sense.

The syntax for passing **dList** and **bList** however is significantly different from the previous examples. What you see is something closely akin to an assignment statement. In other words, the parameter named **dList** is *assigned* to the argument named **listD** . Also, the parameter named **bList** is assigned to the argument named **listB** . Nothing is passed and assigned to the argument named **listC** . Furthermore, the two parameters that are passed through assignment to the named arguments are passed in reverse order relative to the definition of those arguments in the function definition.

Hopefully this explanation along with the code in Listing 4 and the output in Figure 5 will tell you what you need to know about using keyword function arguments.

I also recommend that you create a visualization for the code in Listing 4 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of keyword arguments.

Variable-length arguments

The program in Listing 5 defines a function named **listModifier** with one required argument (**listA**) and a syntax that supports an arbitrary number of arguments (***wxyz**). *(Note the asterisk, *, immediately to the left of **wxyz** .)*

The program in Listing 5 illustrates *variable-length arguments* .

Listing 5 . Illustration of variable-length arguments.

```
# Illustrates variable-length arguments
#-------------------------------------------
----------------------------

def listModifier(listA,*wxyz):
  """Illustrates variable-length arguments"""
  print("In listModifier")

  #append a numeric value to the list
referenced by required argument listA
  listA.append(1.00001)

  #append increasing numeric values to lists
referenced by other parameters
  count = 2
  for ref in wxyz:
    ref.append(1.00001 * count)
    count += 1 #end for loop here

  return #return nothing
#End function definition
```

Listing 5 . Illustration of variable-length arguments.

```
aList = ["ab","cd","ef"]
bList = ["The","old","list"]
cList = ["This old house"]
dList = ["is falling down"]

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))

print("Call listModifier")
listModifier(aList,bList,cList,dList)
print("Back from listModifier")

print("aList = " + str(aList))
print("bList = " + str(bList))
print("cList = " + str(cList))
print("dList = " + str(dList))
```

The code in Listing 5 produces the output shown in Figure 6 .

Figure 6 . Output produced by the code in Listing 5.

Figure 6 . Output produced by the code in Listing 5.

```
aList = ['ab', 'cd', 'ef']
bList = ['The', 'old', 'list']
cList = ['This old house']
dList = ['is falling down']
Call listModifier
In listModifier
Back from listModifier
aList = ['ab', 'cd', 'ef', 1.00001]
bList = ['The', 'old', 'list', 2.00002]
cList = ['This old house', 3.00003]
dList = ['is falling down', 4.00004]
```

The first thing that you should pay attention to is the syntax for defining variable-length arguments in the function named **listModifier** in Listing 5 . The syntax consists of an asterisk (*) followed by an arbitrary argument name. As mentioned earlier, the function definition in Listing 5 consists of a required argument (**listA**) followed by the syntax for a variable-length group of arguments (***wxyz**).

In this program, the calling program must pass a parameter for the required argument and can pass an arbitrary number of parameters following the required argument.

The second thing you should pay attention to is the syntax used to call the function named **listModifier** about six lines up from the bottom of Listing 5 . The function call passes four parameters: **aList** , **bList, cList** , and **dList** , each of which is a reference to a list object.

The reference variable named **aList** is passed as the first parameter to satisfy the required argument named **listA** . Note that it is passed using the same syntax as in Listing 1 , Listing 2 , Listing 3 and Listing 4 . Because it

is a required argument, it must be passed in the correct order in a positional sense.

The remaining three parameters are passed as variable-length arguments. The interpreter wraps them in a tuple and presents the tuple to the code in the body of the function. *(You learned about tuples in the earlier module titled Itse1359-1100-Indexing and Slicing Tuples and several modules following that one.)*

The code in the body of the function uses the parameter passed as the required argument to append a numeric value to the list referred to by that parameter.

Although a tuple is immutable, the list objects referred to by the elements in the tuple are mutable. A **for** loop in the functions iterates from the beginning to the end of the tuple, extracting the references to the lists and appending an increasing numeric value to the end of each list. This is shown in the last four lines of text in Figure 6 .

Note that if the order in which the function processes the arguments is important, the order in which the calling program passes the parameters must match that order because that is the order in which they will be wrapped in the tuple.

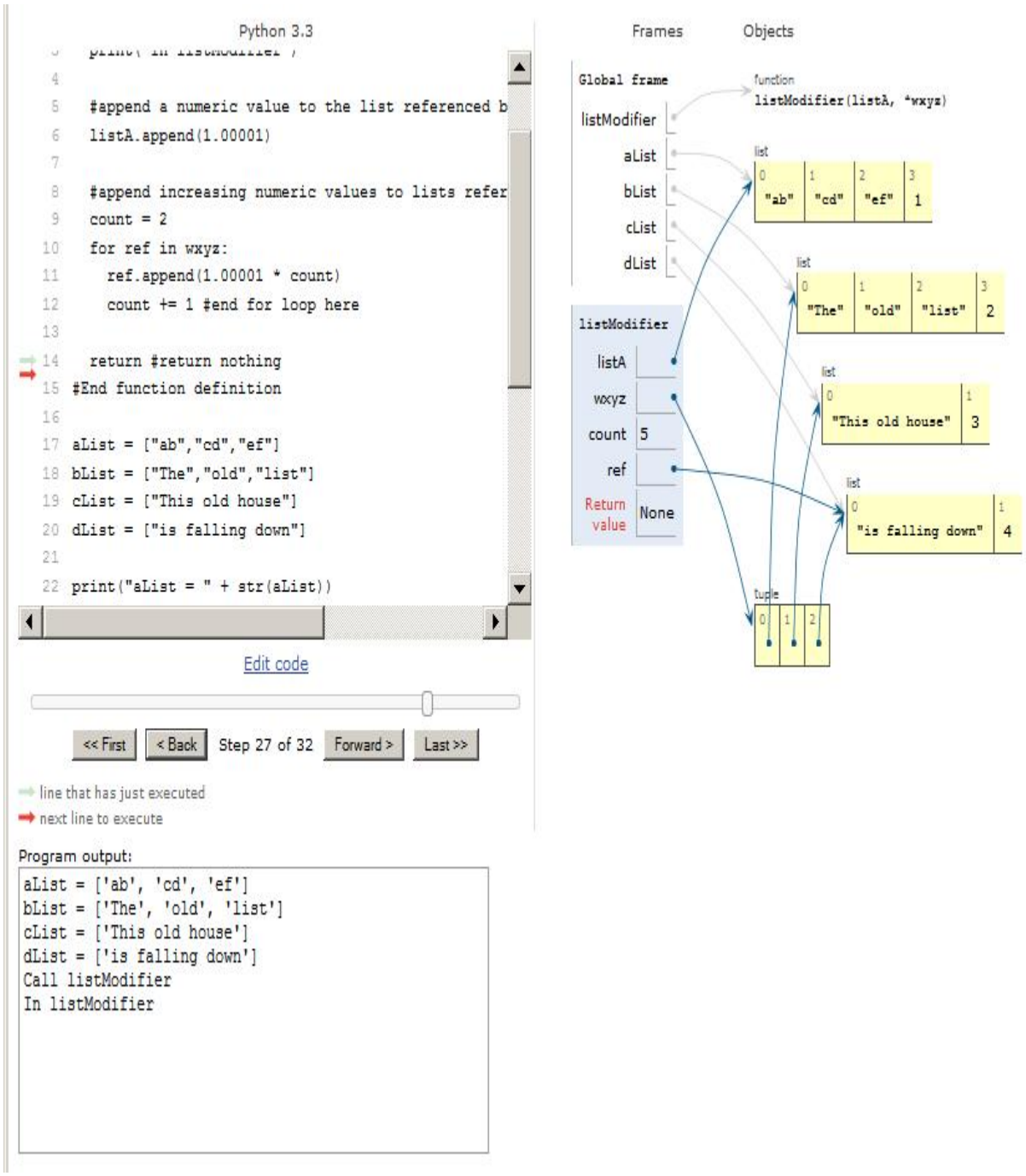
Hopefully this explanation along with the code in Listing 5 and the output in Figure 6 will tell you what you need to know about using variable-length function arguments. I do want to caution again, however, that this module does not provide a complete treatment of function arguments. For a more compete treatment, I will refer you to The Python Tutorial -- More on Defining Functions and tutorialspoint -- Python Functions .

Visualization of variable-length arguments

Figure 7 shows a visualization of the code in Listing 5 part of the way through the execution of the program. Note that in order to reduce the amount of vertical space required to publish the visualization , one of the

visualization parameters was changed to *"inline primitives and nested objects [default]."*

Figure 7. Visualization of variable-length arguments.



I recommend that you create a visualization for the code in Listing 5 and step through the program one instruction at a time. As you do that, pay

attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of variable-length arguments.

Run the programs

I encourage you to copy the code from Listing 1 , Listing 2 , Listing 3 , Listing 4 , and Listing 5 . Execute the code and confirm that you get the same results as those shown in in this module. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

I also recommend that you create visualizations for the code in Listing 1 through Listing 5 . Step through the programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concepts embodied in those sample programs.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1280-Function Arguments
- File: Itse1359-1280.htm
- Published: 10/26/14
- Revised: 01/31/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1410-Overview of Python classes
This is the first of several modules that explain classes and objects in Python. This module provides an overview. Future modules will dig deeper into the details of using classes and objects in Python.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- Preview
- General background information
    - A class definition is a blueprint for objects
    - Three important OOP concepts
        - What is an Object-Oriented Program?
        - What is encapsulation?
- Discussion and sample code
    - A real-world analogy -- a car radio
        - The ability to store data
        - The user interface
        - Modifying the stored data
        - Responding to a message
        - Jargon
        - Where do objects come from?
    - Program code
        - Manufacture a three-button radio
        - Scan for available stations

        - Program the buttons on radio01
        - Play the three programmed stations
        - Manufacture another 3-button radio
        - The class named Radio
        - A class variable named stations
        - The __init__ method
        - The word self
        - Three more methods

- Visualizing a class definition
- Run the program
- Complete program listing
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Previous modules have helped you to learn the basics of programming in Python such as syntax, numbers, variables, strings, scripts, lists, tuples, dictionaries, operators, etc.

Previous modules have also helped you to learn about control flow including **if** statements, **while** loops, **for** loops, functions, function arguments, etc.

What you will learn

The time has come for you to learn about *classes* and *objects* . This is the first of several modules that explain classes and objects in Python.

This module provides an overview. Future modules will dig deeper into the details of using classes and objects.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: Most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Output from the code in Listing 2.
- Figure 2 . Output from the code in Listing 3.
- Figure 3 . Output from the code in Listing 4.
- Figure 4 . Output from the code in Listing 5.
- Figure 5 . Visualizing a class definition.
- Figure 6 . Output from the program in Listing 9.

Listings

- Listing 1 . Manufacture a three-button radio.
- Listing 2 . Scan for available stations.
- Listing 3 . Program the buttons on radio01.
- Listing 4 . Play the three programmed stations.
- Listing 5 . Manufacture another 3-button radio.
- Listing 6 . Beginning of the class named Radio.
- Listing 7 . The __init__ method.
- Listing 8 . Three more methods.
- Listing 9 . Complete program listing.

Preview

This module will concentrate primarily on a discussion of the Python class and objects created using Python classes.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects. For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data. You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

A simple Python program will be presented and explained to illustrate the definition and use of a Python class. This class simulates the manufacture, programming, and use of a car radio.

You will see the definition of a class named **Radio** . You will see how to write code that simulates pressing the *scan* button on the radio to learn about available radio stations in the area. You will see how to write code that simulates the association of a radio button with a particular radio station. You will see how to write code that simulates the pressing of a radio button to play the radio station associated with that button.

You will see how to write code to create new **Radio** objects. You will also see how to save these object's reference in reference variables and how to use those variables to exercise the **Radio** objects.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors. You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects.

You will learn that a Python object is an instance of a class. You will see Python code, used to create two objects from the same class, and then to send messages to each of those objects *(call methods on objects)* .

General background information

A class definition is a blueprint for objects

A class definition is a blueprint *(set of plans)* from which many individual objects can be constructed, created, produced, instantiated, or whatever verb you prefer to use for building something from a set of plans.

An object is a programming construct that encapsulates data and the ability to manipulate that data in a single software entity. The blueprint describes the data contained within and the behavior of objects instantiated according to the class definition.

An object's data is contained in variables defined within the class *(often called member variables, instance variables, data members, attributes, fields, properties, etc.)* . The terminology used in the literature often depends on the background of the person writing the document.

> Note that unlike Java and C++, once a Python object is instantiated, it can be modified in ways that no longer follow the blueprint of the class through the addition of new data members.

An object's behavior is controlled by methods defined within the class. In Python, methods are functions that are defined within a class definition.

An object is said to have *state* and *behavior* . At any instant in time, the state of an object is determined by the values stored in its variables and its behavior is determined by its methods.

Three important OOP concepts

OOP is an abbreviation for Object-Oriented Programming. Most books on OOP will tell you that in order to understand OOP, you need to understand **the following three concepts** :

- Encapsulation
- Inheritance
- Polymorphism

I will discuss the first two concepts in more detail in this and future modules. As near as I can tell, unlike C++ and Java, Python does not support polymorphism, at least not in any significant way.

C++ and Java support two forms of polymorphism:

- Compile-time polymorphism
- Runtime polymorphism

Both of these depend on the "strongly-typed" nature of C++ and Java. Because Python is a "weakly-typed" *(if typed at all)* programming language, I don't know how to implement either form of polymorphism using Python. *(However, if I am wrong on this, please let me know and I will be happy to learn how to implement polymorphism in Python.)*

Generally, speaking, the concepts in the above list increase in difficulty going down the list from top to bottom. Therefore, I will begin with encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program ?

Many authors would answer this question something like the following:

> An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object ?

An object is a software construct that encapsulates data, along with the ability to use or modify that data.

What is encapsulation ?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

> "Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface. The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. I do need to point out that according to The Python Tutorial ,

> "In fact, nothing in Python makes it possible to enforce data hiding -- it is all based upon convention."

Within that restriction, it is still possible to approximate encapsulation as described by Lhotka using Python provided that appropriate conventions are adhered to.

Discussion and sample code

A real-world analogy -- a car radio

Abstract concepts, such as the concept of an object or encapsulation, can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the radio in your car.

The ability to store data

Your car radio probably has the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data through use of the human interface that is provided by the manufacturer of the radio.)* The data that can be stored in your car radio probably includes a list of five or more frequencies that correspond to your favorite radio stations.

> Most modern car radios are much more complex than the one that we will use as an analogy in this module.

The user interface

The radio provides a mechanism *(user interface)* that allows you to use the data stored therein. When you press one of the station-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)* If you have previously stored a favorite radio station frequency in the storage location corresponding to that button, pressing the button *(sending the message)* will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

What I have done in this process is to send a message to the radio object asking it to *change its state* . The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. *(Alternately, we might say that the radio object sent a message back to me.)*

Responding to a message

We say that an object has changed its state when one or more data values stored in the object have been modified. We also say that when an object responds to a message, it will usually

- perform an action,
- change its state,
- return a value, or
- some combination of the above.

After following this procedure to program a button, when I press that button *(send a message)* , the radio object will automatically tune itself to that frequency.

I live in Austin, TX. If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set the frequency value* associated with that button to correspond to one of the radio stations in Dallas. *(Again, I would be sending a message to the radio object asking it to change its state.)*

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of

modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the state of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the station-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action *(tune to a particular station)* . That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Calling a method

Python-speak is a little more specific than general OOP-speak. In Python-speak, we might say that pressing one of the selector buttons on the front of the radio calls a method on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Python is to cause that object to execute one of its methods.

Setter methods and getter methods

Similarly, we might say that storing a new frequency that corresponds to a particular button calls a *setter* method on the radio object. *(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)*

Behavior of an object

In addition to state, objects are often also said to have behavior. The overall behavior of an object is determined by the combined behaviors of its individual methods. For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers. Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from ?

In order to mass-produce car radios, someone must first create a set of manufacturing plans *(drawings, or blueprints)* for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A Python class definition is a set of plans

The same is true for software objects. In order to create a software object in Python, it is necessary for someone to first create a plan. In Python, we refer to that plan as a ***class*** . The class is defined by a Python programmer. Once the class definition is available, that programmer *(or other programmers)* , can use it to produce large numbers of nearly identical objects.

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a new radio and say that it is an instance of the plans used to produce the radio. *(Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.)* However, it is common jargon to refer to a software object as an instance of a class.

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

Program code

As mentioned above, I will explain a program that uses the analogy of a car radio to illustrate several aspects of Python classes and objects.

I will explain this program in fragments. A complete listing of the program is provided in Listing 9 near the end of the module. The output from running the program is shown in Figure 6 . Because this is an overview module, this explanation will gloss over various details. I will revisit and explain many of those details in future modules.

Manufacture a three-button radio

As you will see if you examine Listing 9 , this program defines a class named **Radio** . That class can be used to create objects that simulate car radios having three station-selector buttons and a scan button. I will explain that class later. First I will explain how to use the class.

Listing 1 shows the code necessary to create *(instantiate)* a single instance of the **Radio** class and to store that object's reference in a variable named **radio01** . *(In OOP jargon we say that an object is an instance of a class.)* This simulates the manufacturing of the radio and the installation of the radio in a car.

Listing 1 . Manufacture a three-button radio.

```
#Manufacture a 3-button radio

radio01 = Radio()
```

As a practical matter, the code required to instantiate a Python object looks just like the code required to call a function having the same name as the class from which the object is being instantiated. In my opinion, this causes Python to be a little less readable than Java, which uses a special syntax to create an object.

Scan for available stations

Next we want to program the station-selector buttons to cause our favorite radio stations to be played when we press a button. First, however, we will simulate pressing the *scan* button on the radio to learn about the stations that are available in the area.

Listing 2 uses the reference variable named **radio01** to *call a method* named **scan** belonging to the object referred to by **radio01** . As you will see later when we examine the class definition, the **scan** method requires the name of the city as an incoming parameter and returns a reference to a dictionary that relates frequencies to radio station call signs in that city. The returned reference is stored in the variable named **radio01Stations** .

The dictionary is printed in Listing 2 producing the output shown in Figure 1 .

Listing 2 . Scan for available stations.

```
#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio01Stations = radio01.scan("Austin")
print("Available stations in Austin")
print(radio01Stations)
```

As mentioned above, Figure 1 shows the output produced by the code in Listing 2 .

Figure 1 . Output from the code in Listing 2.

Figure 1 . Output from the code in Listing 2.

```
Available stations in Austin
{93.7: 'KLBJ', 91.7: 'KVRX', 98.1: 'KVET', 95.5:
'KKMJ'}
```

Program the buttons on radio01

This program simulates a radio having three station-selector buttons. The code in Listing 3 uses the object's reference stored in **radio01** to call a method named **setStationNumber** three times in succession on the object to program each of the three buttons to the frequencies shown. *(Note that the buttons are numbered 1, 2, and 3 instead of 0, 1, and 2.)*

Listing 3 . Program the buttons on radio01.

```
print("Program the buttons")
radio01.setStationNumber(1,radio01Stations[91.7])
radio01.setStationNumber(2,radio01Stations[95.5])
radio01.setStationNumber(3,radio01Stations[98.1])
```

The output produced by the code in Listing 3 is shown in Figure 2 .

Figure 2 . Output from the code in Listing 3.

```
Program the buttons
```

Play the three programmed stations

Now that our favorite radio stations in the local area *(Austin)* have been programmed into the buttons, we can play any of the three stations simply by turning the radio on and pressing a station-selector button.

Listing 4 calls the **playStation** method three times in succession on the **radio01** object to simulate pressing each of the three buttons. This program doesn't actually produce sound. Instead, it simulates playing a radio station by printing a message identifying the call sign of the station being played.

Listing 4 . Play the three programmed stations.

```
print("Play the three programmed stations")
radio01.playStation(3)
radio01.playStation(2)
radio01.playStation(1)
```

Figure 3 shows the output produced by the code in Listing 4 .

Figure 3 . Output from the code in Listing 4.

```
Play the three programmed stations
Playing KVET
Playing KKMJ
Playing KVRX
```

Manufacture another 3-button radio

As I explained earlier, once a class definition is available, it can be used to instantiate any number of objects. This is illustrated in Listing 5 , which simulates the manufacturing, programming, and playing of a different radio object.

As you can see in Listing 5 , the buttons on this radio are programmed for stations in Dallas suggesting the owner of this radio lives in Dallas.

Listing 5 . Manufacture another 3-button radio.

Listing 5 . Manufacture another 3-button radio.

```
#Manufacture another 3-button radio
radio02 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio02Stations = radio02.scan("Dallas")
print("Available stations in Dallas")
print(radio02Stations)

print("Program the buttons")
radio02.setStationNumber(1,radio02Stations[91.7])
radio02.setStationNumber(2,radio02Stations[97.9])
radio02.setStationNumber(3,radio02Stations[98.3])

print("Play the three programmed stations")
radio02.playStation(3)
radio02.playStation(2)
radio02.playStation(1)
```

Figure 4 shows the output produced by the code in Listing 5 .

Figure 4 . Output from the code in Listing 5.

Figure 4 . Output from the code in Listing 5.

```
Available stations in Dallas
{98.3: 'KNON', 98.7: 'KLUV', 91.7: 'KKXT', 97.9:
'KBFB'}
Program the buttons
Play the three programmed stations
Playing KNON
Playing KBFB
Playing KKXT
```

The class named Radio

The definition of the class named **Radio** begins in Listing 6 . The first line of code in a class definition consists of the keyword class followed by the name of the class. This is followed by the name of another class in parentheses, which I will explain in a future module on *class inheritance* . As is common in Python, the closing parenthesis is followed by a colon (:) character. The colon is followed by the indented class body.

Listing 6 . Beginning of the class named Radio.

Listing 6 . Beginning of the class named Radio.

```
class Radio(object):
    #This class provides the plans from which the
radio objects are built.
    stations = {"Austin":
{91.7:"KVRX",95.5:"KKMJ",98.1:"KVET",93.7:"KLBJ"},
                "Dallas":
{98.3:"KNON",91.7:"KKXT",97.9:"KBFB",98.7:"KLUV"}
               }
```

A class variable named stations

The class body in Listing 6 begins with a variable named **stations** that references a dictionary object. The dictionary object contains two nested dictionary objects. One of the nested objects provides information about radio stations in Austin. The other nested object provides information about radio stations in Dallas. This is the source of the radio call sign information for Austin and Dallas shown in Figure 3 and Figure 4 .

> IMPORTANT: According to The Python Tutorial -- Class and Instance Variables , the variable named **stations** is a *class variable* that is shared by all instances *(objects)* of the class. I will have more to say about *class variables* and *instance variables* in a future module.

The __init__ method

As I mentioned earlier, when a function is defined inside of a class definition, it is called a method. Listing 7 defines a special method named **__init__** . *(Note that two underscore characters are required on each side of the word **init** .)*

Listing 7 . The __init__ method.

```
    def __init__(self):
        self.stationNumber = [0,0,0]
```

According to The Python Tutorial -- Class Objects ,

> "When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance."

Thus, the **__init__** method can be used to initialize various aspects of a new object when it is instantiated. *(This is roughly analogous to the constructor in other OO languages.)*

The code inside the **__init__** method in Listing 7 creates a new variable named **stationNumber** and initializes it to refer to a three-element list.

> According to The Python Tutorial -- Class and Instance Variables , variables that are created in this manner *(inside class methods)* are called *instance variables* and are *"for data unique to each instance (object)."*

Therefore, the variable named **stations** from Listing 6 is shared among **radio01** and **radio02** in Listing 9 . However, **radio01** and **radio02** each has its own copy of the variable named **stationNumber** from Listing 7 and those variables are not shared among objects.

The word self

Note the use of the word **self** in two locations in Listing 7 . I will explain the use of the word **self** in class definitions in a future module.

Three more methods

Listing 8 defines three more methods. Except for the use of the word **self** , there is nothing unusual about the code in these methods.

Listing 8 . Three more methods.

```
    def scan(self,city):
        return self.stations[city]

    def setStationNumber(self,index,station):
        self.stationNumber[index-1] = station

    def playStation(self,index):
         print("Playing " +
self.stationNumber[index-1])
```

The scan method

The purpose of the method named **scan** is to simulate pressing the *scan* button on the radio. This method is called on the **Radio** object referred to by **radio01** in Listing 2 . It returns a reference to one of the nested dictionary objects in the **stations** dictionary object.

It is called again on a different **Radio** object referred to by **radio02** in Listing 5 . It is important to note that even though both objects were instantiated from the same **Radio** class, they are different objects. The only things they share are the **stations** variable, their ancestry, and their overall structure.

> *The analogy to a physical car radio breaks down with respect to the class variable named **stations** . Physical car radios don't share any data. You could say that data stored in physical car radios is stored in instance variables only.*

The setStationNumber method

The purpose of the **setStationNumber** method is to simulate programming the station-selector buttons on a **Radio** object. It is called three times, once for each button, on the object referred to by **radio01** in Listing 3 . It is also called three times on the different **Radio** object referred to by **radio02** in Listing 5 .

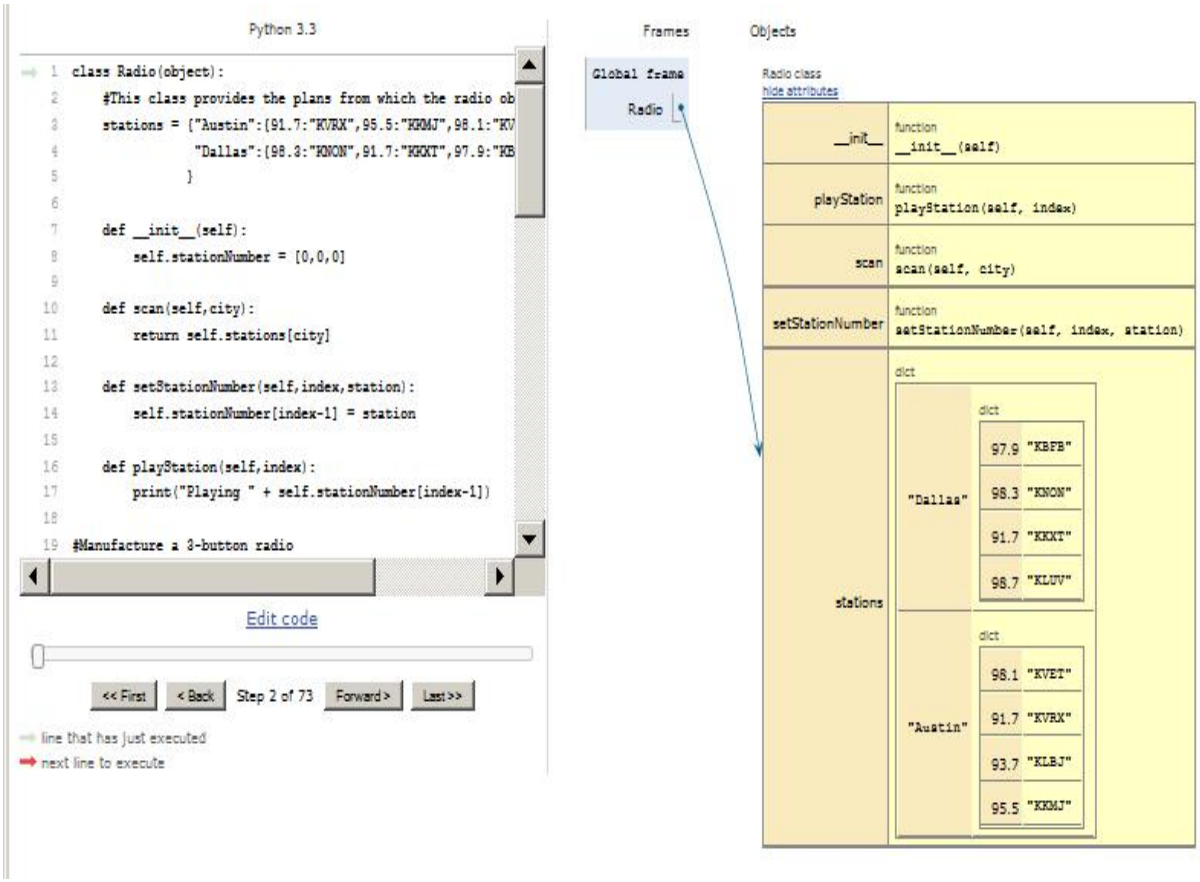
The playStation method

The purpose of the **playStation** method is to simulate pressing a station-selector button on the radio to play the radio station that has been programmed into that button. It is called three times on the **Radio** object referred to by **radio01** in Listing 4 . It is also called three times on the different **Radio** object referred to by **radio02** in Listing 5 .

Visualizing a class definition

Figure 5 shows a visualization of the class definition in Listing 9 .

Figure 5. Visualizing a class definition.



I recommend that you create a visualization for the code in Listing 9 and step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand the concept of classes and objects in Python.

Run the program

I also encourage you to copy the code from Listing 9. Execute the code and confirm that you get the same results as those shown in Figure 6. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is provided in Listing 9 below.

Listing 9 . Complete program listing.

```
# This program simulates the manufacture,
programming, and use of a pair of
# three-button car radios.
#
#-----------------------------------------------
----------------------
class Radio(object):
    #This class provides the plans from which the
radio objects are built.
    stations = {"Austin":
{91.7:"KVRX",95.5:"KKMJ",98.1:"KVET",93.7:"KLBJ"},
                "Dallas":
{98.3:"KNON",91.7:"KKXT",97.9:"KBFB",98.7:"KLUV"}
              }

    def __init__(self):
        self.stationNumber = [0,0,0]

    def scan(self,city):
        return self.stations[city]

    def setStationNumber(self,index,station):
        self.stationNumber[index-1] = station

    def playStation(self,index):
        print("Playing " + self.stationNumber[index-
1])

#Manufacture a 3-button radio
radio01 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio01Stations = radio01.scan("Austin")
print("Available stations in Austin")
print(radio01Stations)

print("Program the buttons")
```

```
radio01.setStationNumber(1,radio01Stations[91.7])
radio01.setStationNumber(2,radio01Stations[95.5])
radio01.setStationNumber(3,radio01Stations[98.1])

print("Play the three programmed stations")
radio01.playStation(3)
radio01.playStation(2)
radio01.playStation(1)



#Manufacture another 3-button radio
radio02 = Radio()

#Program the three buttons labeled 1, 2, and 3
#First scan for available stations
radio02Stations = radio02.scan("Dallas")
print("Available stations in Dallas")
print(radio02Stations)

print("Program the buttons")
radio02.setStationNumber(1,radio02Stations[91.7])
radio02.setStationNumber(2,radio02Stations[97.9])
radio02.setStationNumber(3,radio02Stations[98.3])

print("Play the three programmed stations")
radio02.playStation(3)
radio02.playStation(2)
radio02.playStation(1)
```

Figure 6 shows the output produced by the code in Listing 9 .

Figure 6 . Output from the program in Listing 9.

Figure 6 . Output from the program in Listing 9.

```
Available stations in Austin
{93.7: 'KLBJ', 91.7: 'KVRX', 98.1: 'KVET', 95.5:
'KKMJ'}
Program the buttons
Play the three programmed stations
Playing KVET
Playing KKMJ
Playing KVRX
Available stations in Dallas
{98.3: 'KNON', 98.7: 'KLUV', 91.7: 'KKXT', 97.9:
'KBFB'}
Program the buttons
Play the three programmed stations
Playing KNON
Playing KBFB
Playing KKXT
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1410-Overview of Python classes
- File: Itse1359-1410.htm
- Published: 10/27/14
- Revised: 02/08/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1410r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1410-Overview of Python classes.

Table of contents

- Preface
- Questions
    - 1
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1410-Overview of Python classes* .

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The code in Figure 1 produces the output shown in Figure 2 except that the order of the items in the dictionaries may be different from one run to the next.

Figure 1 . Question 1 program code.

```
class MyNewClass(object):
    myDict = {"A":
{100:"M",200:"N",300:"P",400:"Q"},
              "B":
{500:"S",600:"R",700:"U",800:"T"}
             }

    def __init__(self):
        self.itemNumber = [0,0,0]

    def scan(self,group):
        return self.myDict[group]

    def setItemNumber(self,index,val):
        self.itemNumber[index-1] = val

    def displayItem(self,index):
        print("Showing " +
self.itemNumber[index-1])
```

Figure 1 . Question 1 program code.

```
group01 = MyNewClass()

group01Items = group01.scan("A")
print("Available items in 'A'")
print(group01Items)

group01.setItemNumber(1,group01Items[200])
group01.setItemNumber(2,group01Items[100])
group01.setItemNumber(3,group01Items[300])

group01.displayItem(3)
group01.displayItem(2)
group01.displayItem(1)

group02 = MyNewClass()

group02Items = group02.scan("B")
print("Available items in 'B'")
print(group02Items)

group02.setItemNumber(1,group02Items[600])
group02.setItemNumber(2,group02Items[700])
group02.setItemNumber(3,group02Items[500])

group02.displayItem(3)
group02.displayItem(2)
group02.displayItem(1)
```

Figure 2 . Question 1 possible output.

```
Available items in 'A'
{200: 'N', 300: 'P', 400: 'Q', 100: 'M'}
Showing P
Showing M
Showing N
Available items in 'B'
{600: 'R', 700: 'U', 500: 'S', 800: 'T'}
Showing S
Showing U
Showing R
```

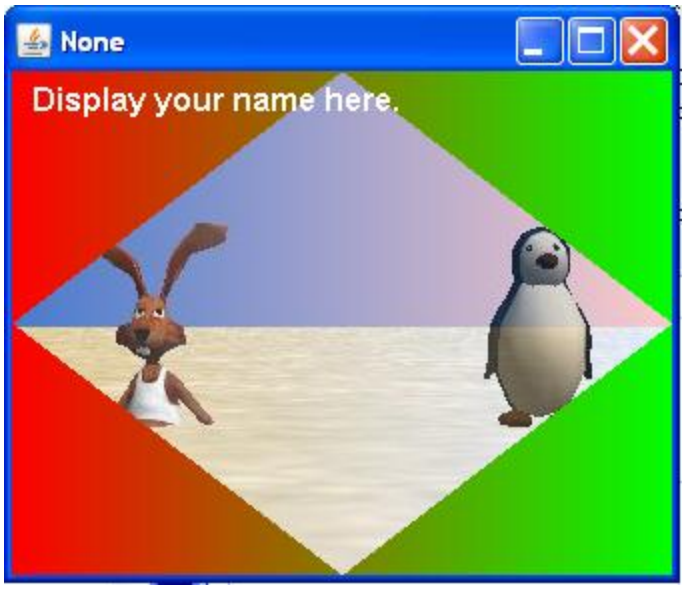
Go to answer 1

Figure index

- Figure 1 . Question 1 program code.
- Figure 2 . Question 1 possible output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 1

True.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1410r-Review
- File: Itse1359-1410r-Review.htm
- Published: 10/27/14
- Revised: 03/02/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1450-Inheritance
You will learn a little about Python class inheritance in this module.

Table of Contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- General background information
    - Inheritance
    - Multiple inheritance
    - Very brief coverage
- Discussion and sample code
    - Define the superclass
    - Define the subclass
    - Exercise the classes
        - Display the value of the class variable
        - Instantiate a Subclass object
        - Modify and then print the value in superclassVar
- Run the program
- Visualize the program
- Complete program listing
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

Earlier modules provided

- an overview of classes and objects in Python,
- an explanation of the **self** word in Python class definitions,
- a detailed explanation of ***instance variables*** in Python class definitions, and
- some of the details regarding ***class variables*** in Python class definitions.

What you will learn

You will learn a little about Python *class inheritance* in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: most of the Figures and all of the Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Output from the code in Listing 3.
- Figure 2 . Output from the code in Listing 4.
- Figure 3 . Output from the code in Listing 5.
- Figure 4 . Output from code in Listing 6.

Listings

- Listing 1 . Define the superclass.
- Listing 2 . Define a subclass that extends or inherits from Superclass.
- Listing 3 . Display the value of the class variable.
- Listing 4 . Instantiate a Subclass object.
- Listing 5 . Modify and then print the value in superclassVar.
- Listing 6 . Complete program listing.

General background information

I told you in an earlier module that OOP is an abbreviation for Object-Oriented Programming. I told you that most books on OOP will tell you that in order to understand OOP, you need to understand **the following three concepts** :

- Encapsulation
- Inheritance
- Polymorphism

I have discussed encapsulation in some detail in previous modules. As near as I can tell, unlike C++ and Java, Python does not support polymorphism, at least not in any significant way.

Inheritance

C++ and Java support two forms of polymorphism:

- Compile-time polymorphism

- Runtime polymorphism

Both of these depend on the "strongly-typed" nature of C++ and Java. Runtime polymorphism using virtual functions and base-class pointers also depends on inheritance.

Because Python is a "weakly-typed" *(if typed at all)* programming language, I don't know how to implement either form of polymorphism using Python. *(However, if I am wrong on this, I will be happy to learn how to implement polymorphism in Python.)*

Inheritance is extremely important in C++ and Java primarily because of the use of runtime polymorphism in program design. However, due to the lack of polymorphism, inheritance in Python is not nearly as important as in those other two languages. Although it may have some other uses, inheritance in Python appears mainly to be a convenient way to reduce the amount of typing required to write a program.

According to tutorialspoint--Python Objected Oriented , Python inheritance is

> "The transfer of the characteristics of a class to other classes that are derived from it."

While that can sometimes be a useful capability, it is simply one of many useful capabilities that exist in Python

Multiple inheritance

C++ supports a concept know as *multiple inheritance* , which often results in significant programming errors for novice programmers. Multiple inheritance is so problematic that the designers of Java excluded it from the language and replaced it with a safer alternative known as the *Java interface* .

Python also supports multiple inheritance. Because of the general lack of safety nets in Python, in my opinion, multiple inheritance is more problematic for novice programmers in Python than in C++. My advice -- don't use Python multiple inheritance for any serious programming effort until you become a very capable Python programmer.

Very brief coverage

As a result of the issues mentioned above, this module will provide only a very brief treatment of inheritance in Python. If you need to learn more about inheritance, I will refer you to the following documents that are available on the web as of October 2014.

- tutorialspoint--Python Objected Oriented
- The Python Tutorial -- Inheritance

Discussion and sample code

I will present and explain a simple program to give you a taste of inheritance. A complete listing of the program is provided in Listing 6 . The output produced by running the program is shown in Figure 4 . As usual, I will break the program down and explain it in fragments.

Define the superclass

Listing 1 defines a class named **Superclass** that will be the *superclass* or the *base class* for another class named **Subclass** . In other words, the class named **Subclass** will *inherit from* the class named **Superclass** or will *extend* the class named **Superclass** , whichever jargon you prefer. *(A visualization of the program uses the word **extends** to indicate inheritance.)*

Depending on your preferred jargon, **Superclass** will be the *base class* or the *superclass* . **Subclass** will be the *derived* class or the *subclass* .

Listing 1 . Define the superclass.

```
#Define the superclass
class Superclass(object):
    superclassVar = True

    def __init__(self,name):
        Superclass.superclassVar = name
        self.friend = "Tom"
```

The code in Listing 1 creates a class variable named **superclassVar** and initializes its value to **True** .

The code in Listing 1 also defines an initializer or constructor method named **__init__** . When this method is executed, it changes the value stored in **superclassVar** from **True** to the value of an incoming parameter named **name** .

The **__init__** method also creates an instance variable named **friend** in the object referred to by **self** and initializes that variable to contain **"Tom"** .

Define the subclass

The class named **Subclass** , which extends **Superclass** , is defined in Listing 2 . This class also defines a constructor method named **__init__** , which in turn calls the **__init__** method of **Superclass** , passing its own incoming parameters to the superclass constructor method.

Listing 2 . Define a subclass that extends or inherits from Superclass.

```
#Define a subclass that extends or inherits
from Superclass
class Subclass(Superclass):
    def __init__(self,name):
        Superclass.__init__(self,name)
        print(self.friend)
```

The method named **__init__** in **Subclass** also prints the value of the variable named **friend** when it is executed.

> When you instantiate a new object of a class that contains a method named __init__, that method is automatically called and a reference to the object is automatically passed as the first parameter to the method. However, if you write code that explicitly calls __init__, you must explicitly pass that parameter along with any other required parameters as shown in Listing 2.

Exercise the classes

The remaining program code is designed to exercise the classes and to illustrate the behavior of inheritance in Python. The code that exercises the classes begins in Listing 3 .

Display the value of the class variable

The code in Listing 3 gets and prints the initial value stored in the class variable named **superclassVar** , producing the output shown in Figure 1 .

Listing 3 . Display the value of the class variable.

```
#Code that exercises the classes
print(Superclass.superclassVar)
```

As you can see, at this point in the program, the class variable contains the value **True** that was placed there when the class variable was created by the code in Listing 1 .

Figure 1 . Output from the code in Listing 3.

```
True
```

Instantiate a Subclass object

The code in Listing 4 instantiates an object of the class named **Subclass** passing **"Joe"** as a parameter to the **__init__** method belonging to that

class. *(As you will recall, that **__init__** method is automatically called when the object is instantiated.)*

Figure 2 shows the output produced by executing the code in Listing 4 .

Listing 4 . Instantiate a Subclass object.

```
ref = Subclass("Joe")

print(ref.superclassVar)
```

As you saw in Listing 2 , the **__init__** method in the class named **Subclass** calls the method named **__init__** in the class named **Superclass** passing both of its incoming parameters as parameters to the **__init__** method in the class named **Superclass** . As you saw in Listing 1 , that method replaces the value **True** in **superclassVar** with one of its incoming parameters, which in this case is **"Joe"** .

The code in the method named **__init__** in the class named **Subclass** in Listing 2 also prints the value of the variable named **friend** *(which is created and initialized by the **__init__** method in the class named **Superclass** in Listing 1)* . This produces the first line of output text *(Tom)* shown in Figure 2 .

The code in Listing 4 also uses the reference to the **Subclass** object to get and print the value of the class variable named **superclassVar** from the class named **Superclass** , producing the second line of output text in Figure 2 *(Joe)* .

Figure 2 . Output from the code in Listing 4.

```
Tom
Joe
```

Modify and then print the value in superclassVar

Finally, the code in Listing 5 uses the reference to the **Subclass** object to modify and then print the value stored in the class variable of the **Superclass** class named **superclassVar** .

Listing 5 . Modify and then print the value in superclassVar.

```
ref.superclassVar = "Sue"
print(ref.superclassVar)
```

Figure 3 shows the output produced by executing the code in Listing 5 .

Figure 3 . Output from the code in Listing 5.

Figure 3 . Output from the code in Listing 5.

```
Sue
```

Run the program

I encourage you to copy the code from Listing 6 . Execute the code and confirm that you get the same results as those shown in Figure 4 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Visualize the program

I also encourage you to create a visualization for the code in Listing 6 . Step through the program one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. That should help you to better understand inheritance in Python.

Complete program listing

A complete listing of the program discussed in this module is provided in Listing 6 below. The output produced by the code in Listing 6 is shown in Figure 4 .

Listing 6 . Complete program listing.

Listing 6 . Complete program listing.

```
# Illustrates inheritance
#-----------------------------------------
-------------
#Define the superclass
class Superclass(object):
    superclassVar = True

    def __init__(self,name):
        Superclass.superclassVar = name
        self.friend = "Tom"

#Define a subclass that extends or inherits
from Superclass
class Subclass(Superclass):
    def __init__(self,name):
        Superclass.__init__(self,name)
        print(self.friend)

#Code that exercises the classes
print(Superclass.superclassVar)
ref = Subclass("Joe")
print(ref.superclassVar)
ref.superclassVar = "Sue"
print(ref.superclassVar)
```

Figure 4 . Output from code in Listing 6.

Figure 4 . Output from code in Listing 6.

```
True
Tom
Joe
Sue
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1450-Inheritance
- File: Itse1359-1450.htm
- Published: 10/27/14
- Revised: 03/04/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1610-GUI Programming
This module explains the ins and outs of creating a simple GUI using Python and tkinter.

Table of Contents

- Preface
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- General background information
    - Tk and tkinter
        - What is a GUI?
        - What is TCL?
        - What is TK?
        - What is tkinter?
    - Not the only GUI alternative
    - A big and complex topic
- Discussion and sample code
    - A sample program
    - Will discuss in fragments
        - The root
        - Import statements
        - The calculate function
        - Add a Frame widget
        - The grid geometry manager
        - Columns and rows
        - A sticky situation

        - StringVar
        - The Entry widget
        - The width argument
        - The textvariable argument
        - Make the Entry field visible
        - A sticky demonstration
        - A Label and a Button
        - Three more Label widgets
        - Finishing touches
        - Now back to the calculate function
        - The input value in feet
        - Either save the float or pass
        - Calculate and display the result in meters
        - Make everything run

- Run the program
- Complete program listing
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module is not intended to be a stand-alone tutorial. Instead, it is intended to be an annotated guide to other freely available online resources on the topic.

What you will learn

You will learn how to create a simple GUI using Python and **tkinter** .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

(Note to blind and visually impaired students: the Figures in this module are bitmap images and are probably not accessible using an audio screen reader or a braille display. The Listings are text and should be accessible. Therefore, this module is only partially accessible to blind students.

Because the module is not accessible to blind students, material from this module will not be used for assignments and will not be included on tests. However, sighted students are strongly encouraged to study the material for their own education. Blind and visually impaired students are also encouraged to learn as much from the module as they can.)

Figures

- Figure 1 . Image of a tkinter test from the command line.
- Figure 2 . Output from Feet to Meters converter program.
- Figure 3 . Output from Listing 1.
- Figure 4 . A sticky demonstration.

Listings

- Listing 1 . The root.
- Listing 2 . Add a Frame widget.
- Listing 3 . StringVar objects.
- Listing 4 . The Entry widget.
- Listing 5 . A Label and a Button.
- Listing 6 . Three more Label widgets.
- Listing 7 . Finishing touches.
- Listing 8 . The calculate function.
- Listing 9 . Complete program listing.

General background information

Tk and tkinter

What is a GUI ?

According to Wikipedia ,

> "In computing, a graphical user interface (GUI... sometimes pronounced "gooey" ...) is a type of interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces ... which require commands to be typed on the keyboard."

You probably interface with most of the programs that you use on a daily basis through a GUI. If you use Python to create programs that will be used by non programmers, you may need to provide GUIs for those programs.

What is TCL?

According to the Tcl Developer Xchange ,

> "Tcl (Tool Command Language) is a very powerful but easy to learn dynamic programming language, suitable for a very wide range of uses, including web and desktop applications, networking, administration, testing and many more. Open source and business-friendly, Tcl is a mature yet evolving language that is truly cross platform, easily deployed and highly extensible."

What is TK?

Also according to the Tcl Developer Xchange ,

> "Tk is a graphical user interface toolkit that takes developing desktop applications to a higher level than conventional approaches. Tk is the standard GUI not only for Tcl, but for many other dynamic languages, and can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more."

What is tkinter?

According to The Python Standard Library -- 25. Graphical User Interfaces with Tk ,

> "Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the tkinter package, and its extension, the tkinter.tix and the tkinter.ttk modules. ... To use tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. tkinter...usually comes bundled with Python."

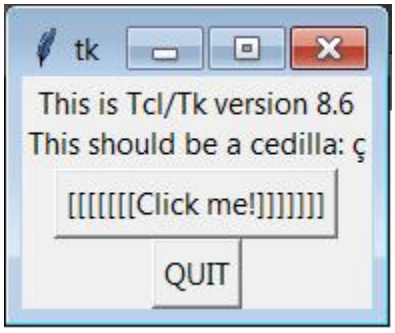
According to The Python Standard Library -- 25.1 tkinter -- Python interface to Tcl/Tk ,

> "The tkinter package ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available

> on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.) You can check that tkinter is properly installed on your system by running python -m tkinter from the command line; this should open a window demonstrating a simple Tk interface."

Figure 1 shows the result of entering the command given above on the command line on my computer and then clicking the button labeled **Click me** several times. Clicking the QUIT button will cause the GUI to disappear from the screen as expected.

Figure 1 . Image of a tkinter test from the command line.



Not the only GUI alternative

You will find other GUI alternatives for Python listed and described at The Python Standard Library -- 25.6. Other Graphical User Interface Packages . However, this module will concentrate on **tkinter** .

A big and complex topic

The design and implementation of graphical user interfaces is a huge topic that could fill an E-book in its own right. In this module, I will barely scratch the surface. In particular, I will present and explain one GUI program. Beyond that, I will refer you to several good online resources

where you can learn more about GUI programming in Python if you are interested.

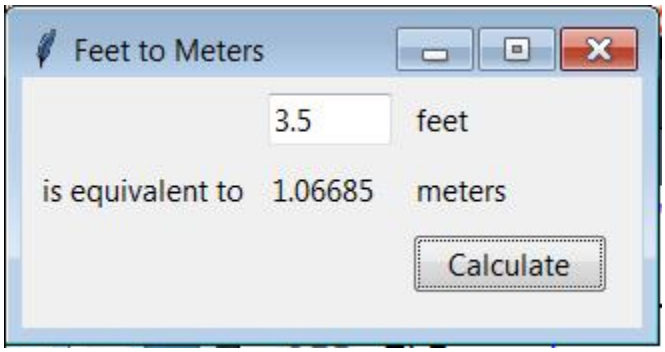
Discussion and sample code

A sample program

Listing 9 presents an unmodified version of a program that was published by Mark Roseman at TkDocs under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Canada (CC BY-NC-SA 2.5 CA) license. This program GUI allows the user to enter a length value in feet. When the user presses a button labeled **Calculate** , the corresponding length value in meters is displayed. This process can be repeated until the user presses the X-button in the upper-right corner of the GUI, which terminates the program and removes the GUI from the screen.

Figure 2 shows the output produced by running this program.

Figure 2 . Output from Feet to Meters converter program.



Will discuss in fragments

As is my custom, I will break the program down and discuss it in fragments. I will attempt to explain why the various parts of the program behave as they do. The first fragment is shown in Listing 1 .

The root

Listing 1 shows a skeleton of the program with most of the code removed to expose the **root** and the *main loop* .

Listing 1 . The root.

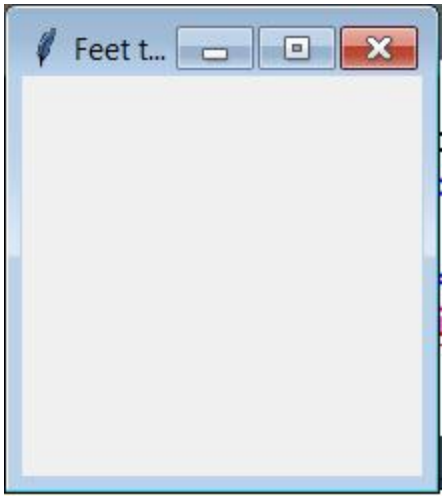
```
from tkinter import *

root = Tk()
root.title("Feet to Meters")

root.mainloop()
```

Figure 3 shows the output produced by running the skeleton code in Listing 1 .

Figure 3 . Output from Listing 1.



According to Hello, Tkinter ,

> "To initialize Tkinter, we have to create a Tk root widget. This is an ordinary window, with a title bar and other decoration provided by your window manager. You should only create one root widget for each program, and it must be created before any other widgets."

*(Note that according to TkDocs , the name of the module was changed from **Tkinter** to **tkinter** with the release of Python 3.0.)*

The empty window that you see in Figure 3 behaves like a standard Windows window on my machine. The three buttons in the upper-right corner behave as expected while the menu icon in the upper-left corner behaves as expected. *(The icon in the upper-left corner appears to be the TCL feather logo.)*

Import statements

TkDocs has this to say about the two import statements at the top of Listing 9 ,

> "These two lines tell Python that our program needs two modules. The first, "tkinter", is the standard binding to Tk, which when loaded also causes the existing Tk library on your system to be loaded. The second, "ttk", is Python's binding to the newer "themed widgets" that were added to Tk in 8.5."

That document goes on to provide a very interesting TIP regarding the use of the asterisk (*) with the first import statement and the non-use of the

asterisk with the second import statement. I will leave it as an exercise for the student to examine that material.

The calculate function

I'm going to skip over the **calculate** function *(see Listing 9)* at this point and return to discuss it in context later.

Add a Frame widget

The first statement in Listing 2 creates a **Frame** widget and places it in the **root** .

Listing 2 . Add a Frame widget.

```
mainframe = ttk.Frame(root, padding="3 3 12
12")
mainframe.grid(column=0, row=0, sticky=(N, W,
E, S))

mainframe.columnconfigure(0, weight=1)
mainframe.rowconfigure(0, weight=1)
```

According to TkDocs ,

> "Next, we create a frame widget, which will hold all the content of our user interface, and place that in our main window. The "columnconfigure"/"rowconfigure" bits just tell Tk that if the main window is resized, the frame should expand to take up the extra space."

According to Tkinter 8.5 reference: a GUI for Python ,

> "Like the Tkinter Frame widget, the ttk.Frame widget is a rectangular container for other widgets."

That document goes on to explain that the first argument to **Frame** is the parent of the **Frame** . In this case, the frame will be a child of **root** . After that, you can enter any of about nine different options, one of which is **padding** .

According to Padding at TkDocs,

> "Normally, each column or row will be directly adjacent to the next, so that widgets will be right next to each other. This is sometimes what you want (think of a listbox and its scrollbar), but often you want some space between widgets. In Tk, this is called padding, and there are several ways you can choose to add it."

One of the ways to achieve the desired space between widgets is by using the syntax shown in the **padding** argument of Listing 2 . This syntax inserts spaces *(in units of pixels)* on the four sides of the frame in the order west, north, east, and south or left, top, right, and bottom. *(Note that the integers in Listing 2 are not separated by commas.)*

The grid geometry manager

Once you have a **Frame** widget, you can call various methods on it. One of those methods is **grid** as shown by the second line of code in Listing 2 .

According to The Grid Geometry Manager at TkDocs,

> "Grid is one of several geometry managers available in Tk, but it's mix of power, flexibility and ease of use, along with its natural fit with today's layouts (that rely on alignment of widgets) make it the best choice for general use. There are other geometry managers: "pack" is also quite powerful, but harder to use and understand; "place" gives you complete control of positioning each element; we'll see even widgets like paned windows, notebooks, canvas and text can act as geometry managers."

Columns and rows

With the **Grid** geometry manager, a container of widgets such as the **root** or the **Frame** is subdivided into a grid of cells. The locations of widgets in the container are specified by column and row numbers.

According to Columns and Rows at TkDocs,

> "Column and row numbers must be integers, with the first column and row starting at 0. You can leave gaps in column and row numbers (e.g. column 0, 1, 2, 10, 11, 12, 20, 21), which is handy if you plan to add more widgets in the middle of the user interface at a later time. The width of each column (or height of each row) depends on the width or height of the widgets contained within the column or row. This means when sketching out your user interface, and dividing it into rows and columns, you don't need to worry about each column or row being equal width."

The second statement in Listing 2 shows that the **Frame** is placed in column 0 and row 0 *(the upper-left corner)* of the **root** .

A sticky situation

The sizes of the cells in a grid are determined by the sizes of the widgets placed in them. Therefore, the width of the cells in a column will be at least as wide as the widest widget in the column and the height of the cells in a row will be at least as high as the tallest widget placed in the row.

By default, if a widget is smaller than the cell in which it is placed, it will be placed in the upper-left corner of the cell. The **sticky** argument can be used to specify something other than the default. The value of the sticky argument consists of the compass directions N, S, E, and W as shown in the second statement of Listing 2 . *(Note that they are separated by commas.)*

Many different combinations of N, S, E, and W can be used to control the placement of the widget in the cell. A good explanation is provided in Layout within the Cell at TkDocs. The combination used in the second statement of Listing 2 causes the **Frame** to be stretched in all four directions so as to be stuck to all four sides of the **root** . In other words, the **Frame** completely fills the available space within the **root** container.

The last two statements in Listing 2 were explained by According to TkDocs above.

StringVar

To make a long story short, the **StringVar** objects shown in Listing 3 are used to connect the string values for **feet** and **meters** to the **textvariable** arguments of an **Entry** widget and a **Label** widget, which you will see later.

*(If you connect a widget to a **StringVar** object using a **textvariable** argument, a change in the contents of either will cause a corresponding change in the other. This is discussed in some detail at 25.1.6.4. Coupling Widget Variables .)*

Listing 3 . StringVar objects.

```
feet = StringVar()
meters = StringVar()
```

The Entry widget

In the initial design stages of the GUI, the author indicates that the widgets will be placed in three columns and three rows.

The first statement in Listing 4 creates a new widget of the class Entry and saves the object's reference in the variable named **feet_entry** . According to The Tkinter Entry Widget ,

> "The Entry widget is a standard Tkinter widget used to enter or display a single line of text."

The first argument is the parent -- in this case **mainframe** . This is followed by many possible optional arguments, one of which is **width** .

Listing 4 . The Entry widget.

```
feet_entry = ttk.Entry(mainframe, width=7,
textvariable=feet)

feet_entry.grid(column=2, row=1, sticky=(W,
E))
```

The width argument

The Tkinter Entry Widget describes the **width** argument as follows:

> "Width of the entry field, in character units. Note that this controls the size on screen; it does not limit the number of characters that can be typed into the entry field. The default width is 20 character."

The first statement in Listing 4 sets the width to seven characters.

The textvariable argument

The Tkinter Entry Widget describes the **textvariable** argument as follows:

> "Associates a Tkinter variable (usually a StringVar) to the contents of the entry field."

This argument is set equal to **feet** from Listing 3 . From this point forward, values entered into the data entry field shown in Figure 2 will be saved in the **StringVar** named **feet** . This makes the value available to the **calculate** function that will be discussed later.

Make the Entry field visible

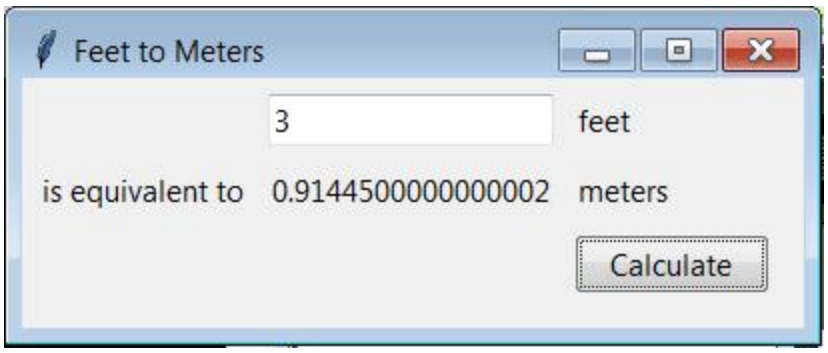
The author of the program explains that the first statement in Listing 4 creates the **Entry** widget but does not make it visible in the GUI. In order to make it visible, you must tell Tk where to place it relative to other widgets. This is accomplished by the call to **grid** in the second statement in Listing 4 , which places the **Entry** widget in column 2 and row 1. *(Note that the author seems to have skipped column 0 and row 0 as described earlier .)*

The **sticky** parameter in Listing 4 instructs Tk to stretch the **Entry** widget horizontally and make it touch both sides of the cell. *(Note that manually stretching the size of the **root** on the screen does not stretch the size of the **Frame** or the widgets contained in the Frame.)*

A sticky demonstration

As an aside, I am going to demonstrate what happens with the sticky parameters in Listing 4 when the column width changes. Compare Figure 4 with Figure 2

Figure 4 . A sticky demonstration.



As you will see shortly, the numerical result of calculating the conversion from feet to meters is displayed in column 2 in the row immediately below the **Entry** widget. When that result requires more width than is available in the column by default, the width of the column is automatically increased. Because the right and left ends of the **Entry** widget stick to the sides of its cell in that column, the width of the **Entry** widget increases accordingly.

A Label and a Button

By now you should be catching on to the general scheme of things and less detailed instructions should be needed.

Listing 5 creates a **Label** widget and a **Button** widget and places them in the rows and columns indicated by the **sticky** parameters.

Listing 5 . A Label and a Button.

```
ttk.Label(mainframe,
textvariable=meters).grid(column=2, row=2,
sticky=(W, E))
ttk.Button(mainframe,text="Calculate",
command=calculate).grid(
    column=3, row=3, sticky=W)
```

A few things are different here:

- References to the widgets are not saved in ordinary Python variables because there will be no need to refer to them later in program code.

- The dot operator is used to call **grid** immediately upon the creation of each widget as opposed to calling **grid** in a separate statement as was done in Listing 4 .
- The **Button** widget uses a **text** argument to place the string **"Calculate"** on the face of the button.
- No text is placed in the **Label** . The empty **Label** will be used to display the results of converting feet to meters.
- The left end of the **Button** widget is stuck to the west side of the cell in which it resides. It is not stretched to the width of the cell as is the case with the **Entry** and **Label** widgets
- Last and perhaps most important, the **Button** widget uses the **command** argument to specify that the function named **calculate** will be executed whenever the user presses the button.

Three more Label widgets

Listing 6 creates three more **Label** widgets and places them in the rows and columns shown.

Listing 6 . Three more Label widgets.

```
ttk.Label(mainframe,
text="feet").grid(column=3, row=1, sticky=W)
ttk.Label(mainframe, text="is equivalent
to").grid(column=1, row=2, sticky=E)
ttk.Label(mainframe,
text="meters").grid(column=3, row=2, sticky=W)
```

There is nothing new in Listing 6 so no explanation of that code should be required.

Finishing touches

The author of the program describes the first line of code in Listing 7 as follows:

> "The first line walks through all of the widgets that are children of our content frame, and adds a little bit of padding around each, so they aren't so scrunched together. We could have added these options to each "grid" call when we first put the widgets onscreen, but this is a nice shortcut."

Listing 7 . Finishing touches.

```
for child in mainframe.winfo_children():
child.grid_configure(padx=5, pady=5)

feet_entry.focus()

root.bind('<Return>', calculate)
```

The author describes the second line of code in Listing 7 as follows:

> "The second line tells Tk to put the focus on our entry widget. That way the cursor will start in that field, so the user doesn't have to click in it before starting to type."

And finally , the author describes the third line of code in Listing 7 as follows:

> "The third line tells Tk that if the user presses the Return key (Enter on Windows) anywhere within the root window, that it should call our calculate routine, the same as if the user pressed the Calculate button."

Now back to the calculate function

The calculate function, which is shown in Listing 8 , is executed whenever the user presses the **calculate** button or hits the *Enter* key on the keyboard as described above .

Listing 8 . The calculate function.

Listing 8 . The calculate function.

```
def calculate(*args):
    try:
        value = float(feet.get())
        meters.set((0.3048 * value * 10000.0 +
0.5)/10000.0)
    except ValueError:
        pass
```

The input value in feet

Any value that is entered into the **Entry** widget is stored in the **StringVar** object referred to by **feet** as shown in Listing 3 and Listing 4 and as described above .

The **StringVar** object has a method named **get** that returns the string stored in the object. The code in the **try** block of Listing 8 calls the **get** method on the **StringVar** object and attempts to convert the string to type **float** .

Either save the float or pass

If the string cannot be converted to a **float** , a **ValueError** is thrown and control is passed to the **pass** statement. This causes the function to exit without doing anything.

If the string can be converted to a **float** , that float value is stored in the variable named **value** .

Calculate and display the result in meters

A **StringVar** object also has a method named **set** that can be used to store a value in the object. The code in the **try** block performs the arithmetic to convert the value in feet to a value in meters and then calls the **set** method on **meters** to store that value in the **StringVar** object referred to by **meters** in Listing 3 .

Storing a value in that **StringVar** object causes the value to appear in the **Label** immediately to the left of the word **meters** in Figure 2 due to the **textvariable** argument in Listing 5 . Once again, if you connect a widget to a **StringVar** object using a **textvariable** argument, a change in the contents of either will cause a corresponding change in the other.

Make everything run

Finally, the last statement in the program, which is shown in Listing 9 , causes Tk to enter its *event-handling loop* and causes everything to run. The code in this loop will monitor for an event indicating that the user has pressed the **calculate** button. When that happens, the **calculate** function will be executed causing the conversion from feet to meters to take place and causing the results of the calculation to be displayed as shown in Figure 2 . This is an infinite loop that will continue to run and monitor for events until the program is terminated.

Run the program

I encourage you to copy the code from Listing 9 . Execute the code and confirm that you get results similar to those shown in Figure 2 or Figure 4 . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Complete program listing

A complete listing of the program is shown in Listing 9 .

Listing 9 . Complete program listing.

```
# Illustrates a Python GUI using tkinter.
# Published by Mark Roseman at TkDocs under a
Creative Commons
# Attribution-NonCommercial-ShareAlike 2.5
Canada (CC BY-NC-SA 2.5 CA) license.
#-----------------------------------------
---------------------------------
from tkinter import *
from tkinter import ttk

def calculate(*args):
    try:
        value = float(feet.get())
        meters.set((0.3048 * value * 10000.0 +
0.5)/10000.0)
    except ValueError:
        pass

root = Tk()
root.title("Feet to Meters")

mainframe = ttk.Frame(root, padding="3 3 12
12")
mainframe.grid(column=0, row=0, sticky=(N, W,
E, S))
mainframe.columnconfigure(0, weight=1)
mainframe.rowconfigure(0, weight=1)

feet = StringVar()
meters = StringVar()

feet_entry = ttk.Entry(mainframe, width=7,
textvariable=feet)
```

Listing 9 . Complete program listing.

```
feet_entry.grid(column=2, row=1, sticky=(W,
E))

ttk.Label(mainframe,
textvariable=meters).grid(column=2, row=2,
sticky=(W, E))
ttk.Button(mainframe,text="Calculate",
command=calculate).grid(
    column=3, row=3, sticky=W)

ttk.Label(mainframe,
text="feet").grid(column=3, row=1, sticky=W)
ttk.Label(mainframe, text="is equivalent
to").grid(column=1, row=2, sticky=E)
ttk.Label(mainframe,
text="meters").grid(column=3, row=2, sticky=W)

for child in mainframe.winfo_children():
child.grid_configure(padx=5, pady=5)

feet_entry.focus()
root.bind('<Return>', calculate)

root.mainloop()
```

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1610-GUI Programming

- File: Itse1359-1610.htm
- Published: 10/28/14
- Revised: 01/31/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1720-Doctest
This module provides an introduction to the doctest module.

Table of contents

- Preface
    - What you have learned
    - What you will learn
    - Viewing tip
        - Figures
        - Listings
- General background information
- Discussion and sample code
    - An interactive session
    - An automated interactive session
        - The test file named Py1359_1720_01
        - The batch the named Py1359_1720_01
        - The output
    - A more substantive example
        - The module named Py1359_1720_02
        - The test file named Py1359_1720_02
        - The batch file named Py1359_1720_02
        - The output
    - Embedding a test in a docstring
        - What is a docstring?
        - The module named Py1359_1720_03
        - The batch file named Py1359_1720_03
        - The output
- Run the program
- What's next?
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

What you have learned

In previous modules, you have learned the basics of Python programming, control flow, classes and objects, input/output, and GUI programming.

What you will learn

In this and the following module, you will learn how to use the Python **doctest** module to test your Python programs.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and the Listings while you are reading about them.

(Note to blind and visually impaired students: all of the Figures and Listings in this module are presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Figures

- Figure 1 . Simple interactive session.
- Figure 2 . Output produced by the test file named Py1359_1720_01.txt.
- Figure 3 . Output produced by the test file named Py1359_1720_02.txt.
- Figure 4 . Output produced by the test embedded in the docstring.

Listings

- Listing 1 . Contents of the test file named Py1359_1720_01.txt.
- Listing 2 . Contents of the file named Py1359_1720_01.bat.
- Listing 3 . Contents of the file named Py1359_1720_02.py.
- Listing 4 . Contents of the test file named Py1359_1720_02.txt.
- Listing 5 . Contents of the batch file named Py1359_1720_02.bat.
- Listing 6 . Contents of the file named Py1359_1720_03.py.
- Listing 7 . Contents of the batch file named Py1359_1720_03.bat.

General background information

According to The Python Standard Library -- 26. Development Tools ,

> "The doctest and unittest modules contain frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced."

The material in this module is based heavily on The Python Standard Library -- 26.2 doctest -- Test interactive Python examples . According to that document,

> "The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown."

Stated differently, the **doctest** module can be used to automate the execution of code that you might otherwise execute in an interactive session. However, instead of having the computer present the result to you, you specify the required result and the computer tests the required result against the actual result and notifies you when the two fail to match.

There are at least **two different ways to use doctest** :

1. You can write the test code in a separate text file and use that file to perform the testing.
2. You can embed the test code in one or more **docstrings** in a module and cause the test code to be executed each time you execute the module as a script.

A simple example of each approach will be provided and explained in this module.

Discussion and sample code

An interactive session

Figure 1 shows a simple interactive session run on the Python interactive command line interface.

Figure 1 . Simple interactive session.

```
>>> 2 + 3
5
>>>
```

If you have forgotten what the text in Figure 1 means, see the earlier module titled *Itse1359-1010-Getting Started* .

An automated interactive session

This first example is provided solely to show how **doctest** works. It is not intended to be useful otherwise.

The test file named Py1359_1720_01

Listing 1 shows the contents of a test file named **Py1359_1720_01.txt** .

Listing 1 . Contents of the test file named Py1359_1720_01.txt.

Listing 1 . Contents of the test file named Py1359_1720_01.txt.

```
>>> 2 + 3
6
```

Note that the contents of the test file mirror the input and the output of the interactive session shown in Figure 1 except that the result of performing the computation was purposely specified incorrectly as 6 instead of 5. *(When you create a **doctest** test file, you must specify the required output as shown in Listing 1 .)*

> Note that the **doctest** process is very picky regarding file names. Some file names that are valid for the operating system may not work when using **doctest** . However, if you restrict your file names to names that would be valid for variables, you should be okay.

Listing 2 shows the contents of a Windows batch file named **Py1359_1720_01.bat** that I used to perform the test. Although not a requirement, the use of such a batch file makes it convenient to perform the test more than once with a minimal typing effort.

The batch the named Py1359_1720_01

Listing 2 . Contents of the file named Py1359_1720_01.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python -m doctest Py1359_1720_01.txt

pause
```

*(Some of the text in Listing 2 is peculiar to my machine. You can ignore the text shown in Listing 2 down to the command that begins with the word **python** .)*

The third command that begins with the word python in Listing 2 executes the Python **doctest** module as a script and passes the name of the test file as a command-line argument to the module.

> According to Python v3.1.5 documentation -- Python Setup and Usage , "When called with -m module-name, the given module is located on the Python module path and executed as a script."

The syntax shown for the python command in Listing 2 is a **command line shortcut** for calling the **testmod()** function in the **doctest** module. *(You will see more on this later.)*

The output

The execution of the python command in Listing 2 produced the command line output shown in Figure 2 .

Figure 2 . Output produced by the test file named Py1359_1720_01.txt.

```
**********************************************************************
File "Py1359_1720_01.txt", line 2, in Py1359_1720_01.txt
Failed example:
2 + 3
Expected:
6
Got:
5
**********************************************************************
1 items had failures:
1 of 1 in Py1359_1720_01.txt
***Test Failed*** 1 failures.
```

Figure 2 shows that the test failed because the result of the computation did not match the specified value of 6. In this case, the output from the computation was correct but the value specified by the test was incorrect.

When testing using the **doctest** module, you specify what the result must be for the test to pass. If the test fails to produce a matching result, the output will show that the test failed. For the case of Listing 1 and Listing 2 , if the correct result had been specified, the test would simply have returned to the command prompt with no other output.

> A verbose output could have been specified in Listing 2 to cause the test to return positive results even if there were no failures. You will see how to do that in the next example.

A more substantive example

The module named Py1359_1720_02

This example will perform a test on a simple function named **sum** in a module named **Py1359_1720_02.py** as shown in Listing 3 .

Listing 3 . Contents of the file named Py1359_1720_02.py.

```
def sum(parA,parB):
    return parA + parB
```

The test file named Py1359_1720_02

The test file for this example is shown in Listing 4 .

Listing 4 . Contents of the test file named Py1359_1720_02.txt.

```
>>> from Py1359_1720_02 import sum
>>> sum(2,3)
6
```

This file will actually cause two tests to be performed. The first test is to import the function named **sum** from the module named **Py1359_1720_02** . However, no visible output is expected from this test. Therefore, no output is specified in Listing 4 .

The second test is to call the function named **sum** *(shown in Listing 3)* passing 2 and 3 as parameters. The test file shows that an incorrect output value of 6 is expected from this test. *(As before, an incorrect value was specified to force the test to fail for illustration purposes only.)*

The batch file named Py1359_1720_02

The batch file used to execute the test is shown in Listing 5 .

Listing 5 . Contents of the batch file named Py1359_1720_02.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python -m doctest -v Py1359_1720_02.txt

pause
```

This batch file differs from the one shown in Listing 2 in one respect. The **-v** switch following **doctest** in the command causes the output to be more verbose than the output from the previous example.

The output

The execution of the third *(python)* command in Listing 5 produced the command line output shown in Figure 3 .

Figure 3 . Output produced by the test file named Py1359_1720_02.txt.

Figure 3 . Output produced by the test file named Py1359_1720_02.txt.

```
Trying:
from Py1359_1720_02 import sum
Expecting nothing
ok
Trying:
sum(2,3)
Expecting:
6
**********************************************************************
File "Py1359_1720_02.txt", line 2, in Py1359_1720_02.txt
Failed example:
sum(2,3)
Expected:
6
Got:
5
**********************************************************************
1 items had failures:
1 of 2 in Py1359_1720_02.txt
2 tests in 1 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

As mentioned earlier, this output is more verbose than the output shown in Figure 2 . In this more verbose output, tests that pass are shown in addition to tests that fail. For example, the test of the **import** statement is shown as **ok** .

On the other hand, as expected, the call to the **sum** function is shown as **failed** . As before, both the location and the nature of the failure is explained.

Embedding a test in a docstring

As mentioned earlier , in addition to putting tests in separate text files, you can also embed tests in docstrings.

What is a docstring?

As shown in the earlier module titled *Itse1359-1270-Functions* , and described more fully in What is a Docstring? , a **docstring** is a string literal that occurs as the first statement in a module, function, class, or method definition. A docstring is surrounded by triple quotes.

The module named Py1359_1720_03

This example will perform a test on a simple function named **sum** in a module named **Py1359_1720_03.py** as shown in Listing 6 .

Listing 6 . Contents of the file named Py1359_1720_03.py.

```
"""
>>> sum(2,3)
6
"""
def sum(parA,parB):
    return parA + parB


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Listing 6 is an update of the earlier module shown in Listing 3 . Test code was embedded in the docstring at the beginning of Listing 6 .

In addition, a special **if** statement was added at the end of Listing 6 that calls the **testmod** function of the **doctest** module. *(See the earlier comment regarding the testmod function and the command line shortcut.)*

According to The Python Standard Library -- 26.2.1. Simple Usage: Checking Examples in Docstrings , **if you include this code** at the end of a module, running the module as a script will cause tests that are embedded in docstrings to be executed.

Detailed information on the various options of the **testmod** function are provided at The Python Standard Library -- 26.2.4. Basic API .

The batch file named Py1359_1720_03

The batch file used to execute the test is shown in Listing 7 .

Listing 7 . Contents of the batch file named Py1359_1720_03.bat.

Listing 7 . Contents of the batch file named Py1359_1720_03.bat.

```
echo off

rem set the path
path=%path%;"C:\Program Files (x86)\Python34"

rem perform the test
python Py1359_1720_03.py -v

pause
```

In this case there was no separate test file. Instead, the python command in Listing 7 executed the module named **Py1359_1720_03.py** as a script triggering the test to be run as described above .

The output

The **-v** switch in Listing 7 causes the verbose version of the output to be produced as shown in Figure 4 . Once again, I set the test up in such a way as to cause it to fail for purposes of illustration.

Figure 4 . Output produced by the test embedded in the docstring.

Figure 4 . Output produced by the test embedded in the docstring.

```
Trying:
sum(2,3)
Expecting:
6
**********************************************************************
File "Py1359_1720_03.py", line 4, in __main__
Failed example:
sum(2,3)
Expected:
6
Got:
5
1 items had no tests:
__main__.sum
**********************************************************************
1 items had failures:
1 of 1 in __main__
1 tests in 2 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Run the program

I encourage you to copy the code provided in this module. Execute the code and confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

What's next?

Now that you know some of the mechanics of using **doctest** , you should study the next module in the collection titled *Itse1359-1720r-Review* . In addition, you should study some of the many articles and tutorials that are freely available on the web that provide insight into the use of **doctest** for unit testing. Links to a few of those documents are provided below. You can find many more with a simple web search.

- 26.2. doctest -- Test interactive Python examples
- Excerpt from book -- Testing in Python using doctest
- Excerpt from book -- Python: Unit Testing with Doctest
- Testing Your Code -- The Hitchhiker's Guide to Python
- Agile Testing -- Python unit testing part 1: the unittest module
- Agile Testing -- Python unit testing part 2: the doctest module
- Agile Testing -- Python unit testing part 3: the py.test tool and library
- Python3 Tutorial: Tests, DocTests, UnitTests -- Python Course
- An Extended Introduction to the nose Unit Testing Framework -- Running doctests in nose
- Doctests: run doctests with nose -- nose 1.3.4 documentation
- Unit Testing Python Using doctest - YouTube

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1720-Doctest Introduction
- File: Itse1359-1720.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1720r-Review
This module contains review questions and answers keyed to the module titled Itse1359-1720-Doctest as well as review questions and answers keyed to the document at https://docs.python.org/3/library/doctest.html.

Table of contents

- Preface
- Questions
    - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to the module titled *Itse1359-1720-Doctest* . In addition, some of the questions and answers are keyed to the document titled The Python Standard Library -- 26.2 doctest -- Test interactive Python examples .

Once you study that material, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The test file shown in Figure 1 , when executed using the command shown in Figure 2 , produces the output shown in Figure 3 .

Figure 1 . Question 1 test code named Py1359_1720r_01.txt.

Figure 1 . Question 1 test code named Py1359_1720r_01.txt.

```
Beginning of first test
>>> 2 + 3
5

End of first test
Beginning of second test
>>> 2 + 3
6

End of second test
```

Figure 2 . Question 1 execution command.

```
python -m doctest Py1359_1720r_01.txt
```

Figure 3 . Question 1 possible output.

Figure 3 . Question 1 possible output.

```
**********************************************************************
File "Py1359_1720r_01.txt", line 2, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
**********************************************************************
File "Py1359_1720r_01.txt", line 7, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
**********************************************************************
1 items had failures:
   2 of   2 in Py1359_1720r_01.txt
***Test Failed*** 2 failures.
```

Go to answer 1

Question 2

True or False? The test file shown in Figure 5 , when executed using the command shown in Figure 6 , produces the output shown in Figure 7 .

Figure 5 . Question 2 test code named Py1359_1720r_02.txt.

```
Beginning of test
>>> 2 + 3
5
End of test
```

Figure 6 . Question 2 execution command.

```
python -m doctest Py1359_1720r_02.txt -v
```

Figure 7 . Question 2 possible output.

```
Trying:
    2 + 3
Expecting:
    5
ok
1 items passed all tests:
   1 tests in Py1359_1720r_02.txt
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
```

Go to answer 2

Question 3

True or False? Given the Python module shown in Figure 9 , when the test file shown in Figure 10 is executed using the command shown in Figure 11 , it produces the output shown in Figure 12 .

Figure 9 . Question 3 Python module named Py1359_1720r_03.py.

```
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD
```

Figure 10 . Question 3 test code named Py1359_1720r_03.txt.

```
>>> from Py1359_1720r_03 import sum
>>> sum(2,3)
5
>>> from Py1359_1720r_03 import product
>>> product(2,3)
6
```

Figure 11 . Question 3 execution command.

```
python -m doctest -v Py1359_1720r_03.txt
```

Figure 12 . Question 3 possible output.

Figure 12 . Question 3 possible output.

```
Trying:
    from Py1359_1720r_03 import sum
Expecting nothing
ok
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    from Py1359_1720r_03 import product
Expecting nothing
ok
Trying:
    product(2,3)
Expecting:
    6
ok
1 items passed all tests:
   4 tests in Py1359_1720r_03.txt
4 tests in 1 items.
4 passed and 0 failed.
Test passed.
```

Go to answer 3

Question 4

True or False? Given the Python module shown in Figure 13 , execution of the module as a script using the command shown in Figure 14 produces the output shown in Figure 15 .

Figure 13 . Question 4 Python module named Py1359_1720r_04.py.

Figure 13 . Question 4 Python module named Py1359_1720r_04.py.

```
"""
Begin first test
>>> sum(2,3)
5

Begin second test
>>> product(2,3)
6
"""
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 14 . Question 4 execution command.

```
python Py1359_1720r_04.py -v
```

Figure 15 . Question 4 possible output.

Figure 15 . Question 4 possible output.

```
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    product(2,3)
Expecting:
    6
ok
2 items had no tests:
    __main__.product
    __main__.sum
1 items passed all tests:
   2 tests in __main__
2 tests in 3 items.
2 passed and 0 failed.
Test passed.
```

Go to answer 4

Question 5

True or False? Given the Python module shown in Figure 16 , execution of the module as a script using the command shown in Figure 17 produces the output shown in Figure 18 .

Figure 16 . Question 5 Python module named Py1359_1720r_05.py.

Figure 16 . Question 5 Python module named Py1359_1720r_05.py.

```
"""
>>> divide(2,3)
0
>>> divide(2,0)
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
"""
def divide(parA,parB):
    return parA//parB

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 17 . Question 5 execution command.

```
python Py1359_1720r_05.py -v
```

Figure 18 . Question 5 possible output.

Figure 18 . Question 5 possible output.

```
Trying:
    divide(2,3)
Expecting:
    0
ok
Trying:
    divide(2,0)
Expecting:
    Traceback (most recent call last):
      ...
    ZeroDivisionError: integer division or modulo by zero
ok
1 items had no tests:
    __main__.divide
1 items passed all tests:
   2 tests in __main__
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Go to answer 5

Question 6

True or False? Given the Python module shown in Figure 19 , execution of the module as a script using the command shown in Figure 20 produces the output shown in Figure 21 .

Figure 19 . Question 6 Python module named Py1359_1720r_06.py.

Figure 19 . Question 6 Python module named Py1359_1720r_06.py.

```
"""
>>> function(11)
Traceback (most recent call last):
  ...
ValueError: par must be <= 10
"""
def function(par):
    if par > 10:
        raise ValueError("par must be <= 10")
    else:
        return par

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 20 . Question 6 execution command.

```
python Py1359_1720r_06.py -v
```

Figure 21 . Question 6 possible output.

Figure 21 . Question 6 possible output.

```
Trying:
    function(11)
Expecting:
    Traceback (most recent call last):
      ...
    ValueError: par must be <= 10
ok
1 items had no tests:
    __main__.function
1 items passed all tests:
   1 tests in __main__
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Go to answer 6

Question 7

True or False? Given the Python modules shown in Figure 22 and Figure 23 and the test file shown in Figure 25 , execution of the command shown in Figure 24 , produces the output shown in Figure 26 .

Figure 22 . Question 7 Python module named Py1359_1720r_07_runner.py.

```
import doctest
doctest.testfile(Py1359_1720r_07.txt)
```

Figure 23 . Question 7 Python module named Py1359_1720r_07.py.

Figure 23 . Question 7 Python module named Py1359_1720r_07.py.

```
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD
```

Figure 24 . Question 7 execution command.

```
python Py1359_1720r_07_runner.py
```

Figure 25 . Question 7 test code named Py1359_1720r_07.txt.

```
>>> from Py1359_1720r_07 import sum
>>> sum(2,3)
5
>>> from Py1359_1720r_07 import product
>>> product(2,3)
5
```

Figure 26 . Question 7 possible output.

Figure 26 . Question 7 possible output.

```
**********************************************************************
File "Py1359_1720r_07.txt", line 5, in Py1359_1720r_07.txt
Failed example:
    product(2,3)
Expected:
    5
Got:
    6
**********************************************************************
1 items had failures:
   1 of   4 in Py1359_1720r_07.txt
***Test Failed*** 1 failures.
```

Go to answer 7

Question 8

True or False? Given the Python module shown in Figure 27 and the test file shown in Figure 28 , execution of the module as a script using the command shown in Figure 29 , produces the output shown in Figure 30 .

Figure 27 . Question 8 Python module named Py1359_1720r_08.py.

```
def sum(parA,parB):
    return parA + parB

def product(parC,parD):
    return parC * parD

if __name__ == "__main__":
    import doctest
    doctest.testfile("Py1359_1720r_08.txt")
```

Figure 28 . Question 8 test code named Py1359_1720r_08.txt.

```
>>> from Py1359_1720r_08 import sum
>>> sum(2,3)
5
>>> from Py1359_1720r_08 import product
>>> product(2,3)
6
```

Figure 29 . Question 8 execution command.

```
python Py1359_1720r_08.py -v
```

Figure 30 . Question 8 possible output.

Figure 30 . Question 8 possible output.

```
Trying:
    from Py1359_1720r_08 import sum
Expecting nothing
ok
Trying:
    sum(2,3)
Expecting:
    5
ok
Trying:
    from Py1359_1720r_08 import product
Expecting nothing
ok
Trying:
    product(2,3)
Expecting:
    6
ok
1 items passed all tests:
   4 tests in Py1359_1720r_08.txt
4 tests in 1 items.
4 passed and 0 failed.
Test passed.
```

Go to answer 8

Question 9

True or False? Given the Python version 3 module shown in Figure 31 , execution of the module as a script using the command shown in Figure 32 , produces the output shown in Figure 33 .

Figure 31 . Question 9 Python module named Py1359_1720r_09.py.

Figure 31 . Question 9 Python module named Py1359_1720r_09.py.

```
"""
>>> divide(1,3)
0.333333
"""
def divide(parA,parB):
    return parA/parB

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Figure 32 . Question 9 execution command.

```
python Py1359_1720r_09.py -v
```

Figure 33 . Question 9 possible output.

```
Trying:
    divide(1,3)
Expecting:
    0.3333333333333333
ok
1 items had no tests:
    __main__.divide
1 items passed all tests:
   1 tests in __main__
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

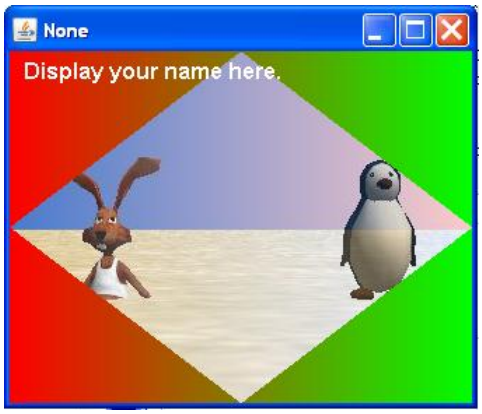
Go to answer 9

Figure index

- Figure 1 . Question 1 test code named Py1359_1720r_01.txt.
- Figure 2 . Question 1 execution command.
- Figure 3 . Question 1 possible output.
- Figure 4 . Question 1 actual output.
- Figure 5 . Question 2 test code named Py1359_1720r_02.txt.
- Figure 6 . Question 2 execution command.
- Figure 7 . Question 2 possible output.
- Figure 8 . Question 2 actual output.
- Figure 9 . Question 3 Python module named Py1359_1720r_03.py.
- Figure 10 . Question 3 test code named Py1359_1720r_03.txt.
- Figure 11 . Question 3 execution command.
- Figure 12 . Question 3 possible output.
- Figure 13 . Question 4 Python module named Py1359_1720r_04.py.
- Figure 14 . Question 4 execution command.
- Figure 15 . Question 4 possible output.
- Figure 16 . Question 5 Python module named Py1359_1720r_05.py.
- Figure 17 . Question 5 execution command.
- Figure 18 . Question 5 possible output.
- Figure 19 . Question 6 Python module named Py1359_1720r_06.py.
- Figure 20 . Question 6 execution command.
- Figure 21 . Question 6 possible output.
- Figure 22 . Question 7 Python module named Py1359_1720r_07_runner.py.
- Figure 23 . Question 7 Python module named Py1359_1720r_07.py.
- Figure 24 . Question 7 execution command.
- Figure 25 . Question 7 test code named Py1359_1720r_07.txt.
- Figure 26 . Question 7 possible output.
- Figure 27 . Question 8 Python module named Py1359_1720r_08.py.
- Figure 28 . Question 8 test code named Py1359_1720r_08.txt.
- Figure 29 . Question 8 execution command.
- Figure 30 . Question 8 possible output.
- Figure 31 . Question 9 Python module named Py1359_1720r_09.py.
- Figure 32 . Question 9 execution command.
- Figure 33 . Question 9 possible output.
- Figure 34 . Question 9 actual output.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 9

False. The actual output is shown in Figure 34 . This question was included to point out that testing floating point results can be problematic. Many floating point values are simply estimates. Although this example was purposely caused to fail, a given floating point computation may not produce exactly the same results on different platforms. Therefore a test that specifies a given floating point result may pass on one platform and fail on a different platform. The Python Standard Library -- 26.2 doctest -- Test interactive Python examples provides some suggestions for dealing with this issue.

Figure 34 . Question 9 actual output.

Figure 34 . Question 9 actual output.

```
Trying:
    divide(1,3)
Expecting:
    0.333333
**********************************************************************
File "Py1359_1720r_09.py", line 4, in __main__
Failed example:
    divide(1,3)
Expected:
    0.333333
Got:
    0.3333333333333333
1 items had no tests:
    __main__.divide
**********************************************************************
1 items had failures:
   1 of   1 in __main__
1 tests in 2 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Go back to Question 9

Answer 8

True. The call to **testfile** at the end of Figure 27 illustrates another way to cause a test file to be executed.

Go back to Question 8

Answer 7

False. The output produced by the procedure described in this question includes the following error message:

NameError: name 'Py1359_1720r_07' is not defined

This indicates a syntax error in the specification of the name of the test file. While the concept is correct and it is possible to use Python code in a module similar to that shown in Figure 22 to execute a test file such as that shown in Figure 25 , the name of the test file must be enclosed in quotation marks to be recognized as a test file. Enclosing the name of the test file in Figure 22 in quotes would cause the procedure to produce the output shown in Figure 26 .

Go back to Question 7

Answer 6

True.

Go back to Question 6

Answer 5

True.

Go back to Question 5

Answer 4

True.

Go back to Question 4

Answer 3

True.

Go back to Question 3

Answer 2

False. The actual output is shown in Figure 8 . This is a formatting issue. If you include explanatory text in the text file, you must leave a blank line between the last line of specified output and the following line of explanatory text. Otherwise, the explanatory text will be included as part of required output as shown by the expected output value in Figure 8 .

Figure 8 . Question 2 actual output.

Figure 8 . Question 2 actual output.

```
Trying:
    2 + 3
Expecting:
    5
    End of test
**********************************************************************
File "Py1359_1720r_02.txt", line 2, in Py1359_1720r_02.txt
Failed example:
    2 + 3
Expected:
    5
    End of test
Got:
    5
**********************************************************************
1 items had failures:
   1 of   1 in Py1359_1720r_02.txt
1 tests in 1 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Go back to Question 2

Answer 1

False. The actual output is shown in Figure 4 . The test file contained two tests, one of which passed and the other of which failed. Verbose output was not specified. Therefore, the report didn't include information on the test that passed. The test file contained explanatory text, which is okay under certain formatting conditions, which I will get into in a subsequent question..

Figure 4 . Question 1 actual output.

Figure 4 . Question 1 actual output.

```
**********************************************************************
File "Py1359_1720r_01.txt", line 7, in Py1359_1720r_01.txt
Failed example:
    2 + 3
Expected:
    6
Got:
    5
**********************************************************************
1 items had failures:
   1 of   2 in Py1359_1720r_01.txt
***Test Failed*** 1 failures.
```

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1720r-Review
- File: Itse1359-1720r.htm
- Published: 11/10/14
- Revised: 03/04/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1900-Preface to Networking and Databases
This module is the preface for the Networking and Databases portion of the course.

Table of contents

- Preface
- Networking with HTTP
- Dbm and shelve databases
- SQLite database
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

The learning resource for the **Networking and Databases** portion of the course consists of material from selected websites plus modules containing review questions and answers keyed to the material on those websites.

The **Networking and Databases** portion of the course consists of three main areas of study:

- Networking with HTTP
- Dbm and shelve databases
- SQLite database

A list of selected websites plus review questions keyed to those websites is provided for each area of study. The combined list of selected websites is:

- tutorialspoint - HTTP - Quick Guide
- w3.org - 3 Protocol Parameters
- The Python Standard Library - 21.12. http.client - HTTP protocol client

- The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"
- The Python Standard Library - 12.3. shelve - Python object persistence
- The Python Standard Library - 12.1. pickle - Python object serialization
- The Python Standard Library - 4.10. Mapping Types - dict
- W3Schools SQL Tutorial
- The Python Standard Library - 12.6. SQLite3 - DB-API 2.0 interface for SQLite databases
- Python Central - Introduction to SQLite in Python
- udemy - SQLite for Beginners
- tutorialspoint - SQLite Tutorial
- ZetCode - SQLite tutorial

Take-home programming assignments may be based on any material on these websites as well as material from any of the other modules in this collection of modules for teaching ITSE 1359. However, online tests for the **Networking and Databases** portion of the course will be restricted to topics addressed in the following three modules:

- Itse1359-1910-Networking with HTTP
- Itse1359-1930-Dbm and Shelve Databases
- Itse1359-1950-SQLite Database

Networking with HTTP

Networking is a very broad topic. There are many textbooks and websites that discuss Python programs used for a variety of networking purposes, such as online chat, shared whiteboards, FTP, sockets, server sockets, etc. However, in order to narrow the topic somewhat, this course and the topics covered by the module titled *Networking with HTTP* concentrate mainly on what is probably the most common form of networking: HTTP.

According to tutorialspoint - HTTP - Quick Guide ,

> The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e.. internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extension of its request methods, error codes and headers.

In other words, if you are reading this document online, you are probably accessing and reading it from a website using HTTP.

Dbm and shelve databases

In layman's terms, a database is a structure that can maintain data on a persistent basis. In other words, the data does not go away when the program that is accessing that data terminates. The same or other programs can be started up later to access and manipulate the data in the database.

Python supports several different types of databases, one of which is DBM . However, configuring a system to support a particular type of database can be very tedious. For compatibility with different student's computing systems, this course will concentrate on a standard Portable DBM implementation known as dbm.dumb.

According to Portable DBM implementation , the **dbm.dumb** module is intended as a last resort fallback for the dbm module when a more robust module is not available. The **dbm.dumb** module is not written for speed and is not nearly as heavily used as the other database modules.

The **dbm.dumb** module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as dbm.gnu no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

A shelf is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

Aside from this brief description, this course will not address shelve .

SQLite database

SQL stands for **Structured Query Language** . SQL is used to communicate with a database. According to ANSI *(American National Standards Institute)* , it is the standard language for relational database management systems.

According to The Python Standard Library - 12.6. sqlite3 - DB-API 2.0 interface for SQLite databases ,

> SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.

SQL is also a very broad topic. Considerable time, effort, and experience is required to master SQL. As evidence of this fact, see the somewhat voluminous W3Schools SQL Tutorial .

Knowledge of SQL is a prerequisite for understanding SQLite. It is unreasonable to expect a student to learn SQL during a small portion of a one-semester course in Python. Therefore, the module titled *Itse1359-1950-SQLite Database* will barely scratch the surface and will provide only a cursory introduction to the use of SQLite.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1900-Preface to Networking and Databases
- File: Itse1359-1900.htm
- Published: 11/11/14
- Revised: 06/23/17

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-1930-Dbm and Shelve Databases
This module contains review questions and answers keyed to instructional material on several websites.

Table of contents

- Preface
- Questions
  - 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15
- Figure index
- Answers
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This module contains review questions and answers keyed to material on the following web sites:

- The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases"
- The Python Standard Library - 12.3. shelve - Python object persistence
- The Python Standard Library - 12.1. pickle - Python object serialization
- The Python Standard Library - 4.10. Mapping Types - dict

Once you study that module, you should be able to answer the review questions in this module.

The questions and the answers in this module are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

(Note to blind and visually impaired students: with the exception of two bitmap images that are used solely as spacers to separate the question section from the answer section, all of the material in this module is presented in plain text format and should be accessible using an audio screen reader or a braille display. Note however that the required indentation may not be properly represented by an audio screen reader.)

Questions

Question 1

True or False? The webpage at The Python Standard Library - 12.1. pickle - Python object serialization contains the following warning:

> Warning: The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Go to answer 1

Question 2

True or False? The webpage at The Python Standard Library - 12.3. shelve - Python object persistence contains the following warning:

> Warning: Because the shelve module is backed by pickle, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Go to answer 2

Question 3

True or False? The webpage at The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases" contains the following warning:

> Warning: The dbm module is not intended to be secure against erroneous or maliciously constructed data. Never use dbm data received from an untrusted or unauthenticated source.

Go to answer 3

Question 4

True or False? **dbm** is a generic interface to variants of the DBM databases - **dbm.gnu** or **dbm.ndbm** . If neither of these modules is installed, the program will raise a "Module not found" error.

Go to answer 4

Question 5

True or False? The function named **dbm.whichdb(*filename*)** can be called to determine which database module should be used to open a given database file.

Go to answer 5

Question 6

True or False? The function named **dbm.getDatabase** can be called to open an existing **dbm** database file or to create a new **dbm** database file.

Go to answer 6

Question 7

True or False? The object returned by **dbm.open** supports the same basic functionality as tuples. Keys and their corresponding values can be stored, retrieved, and deleted. The **in** operator and the k **eys** method are available, as well as **get** and **setdefault** .

Go to answer 7

Question 8

True or False? When using the **dbm.dumb** module, attempting to store a key or value that is not a string will raise an exception.

Go to answer 8

Question 9

True or False? The following statement appears on the webpage at The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases" .

> Note: The dbm.dumb module is intended as a last resort fallback for the dbm module when a more robust module is not available. The dbm.dumb module is not written for speed and is not nearly as heavily used as the other database modules.

Go to answer 9

Question 10

True or False? The function named **dbm.dumb.open** can be called to create or open a **dumbdbm** database and return a **dumbdbm** object.

Go to answer 10

Question 11

True or False? The code in the top panel of Figure 1 produces the output shown in the bottom panel of Figure 1 .

Figure 1 . Question 11.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

dumbDb[b'aNumber'] = 1234
print(dumbDb.get('aNumber'))

dumbDb.close()
================================================================================
1234
```

Go to answer 11

Question 12

True or False? The code in the top panel of Figure 3 produces the output shown in the bottom panel of Figure 3 .

Figure 3 . Question 12.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

dumbDb[b'aNumber'] = b'1234'
print(dumbDb.get('aNumber'))

dumbDb.close()
================================================================================
b'1234'
```

Go to answer 12

Question 13

True or False? The code in the top panel of Figure 4 produces the output shown in the bottom panel of Figure 4 .

Figure 4 . Question 13.

Figure 4 . Question 13.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

empName = b'Joe'

dumbDb[b'name'] = empName
dumbDb[b"cubicle"] = b"639"
dumbDb[b"phone"] = b"617.665.2154"

if(empName in dumbDb):
    print('name: ', dumbDb.get(b'name'))
    print('cubicle: ', dumbDb.get(b'cubicle'))
    print('phone: ', dumbDb.get(b'phone'))
else:
    print("Not in dumbDb")

dumbDb.close()
```

```
================================================================================
name: b'Joe'
cubicle: b'639'
phone: b'617.665.2154'
```

Go to answer 13

Question 14

True or False? The code in the top panel of Figure 6 produces the output shown in the bottom panel of Figure 6 .

Figure 6 . Question 14.

Figure 6 . Question 14.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

empName = b'Joe'

dumbDb[b'name'] = empName
dumbDb[b"cubicle"] = b"639"
dumbDb[b"phone"] = b"617.665.2154"

if(b'name' in dumbDb):
    print('name: ', dumbDb.get(b'name'))
    print('cubicle: ', dumbDb.get(b'cubicle'))
    print('phone: ', dumbDb.get(b'phone'))
else:
    print("Not in dumbDb")

dumbDb.close()
================================================================================
name:  b'Joe'
cubicle:  b'639'
phone:  b'617.665.2154'
```

Go to answer 14

Question 15

True or False? The code in the top panel of Figure 7 produces the output shown in the bottom panel of Figure 7 .

Figure 7 . Question 15.

```
import dbm.dumb

dumbDb = dbm.dumb.open('database', 'c')

#Create keys and values
myKeys = b'name',b'cubicle',b'phone',b'age',b'gender',b'marital'
myValues = b'Joe',b'639',b'333.444.5555',b'39',b'male',b'married'

#Populate the database
for cnt in range(len(myKeys)):
    dumbDb[myKeys[cnt]] = myValues[cnt]

print('Get and display values')
values = dumbDb.values()
for value in values:
    print(value)
```

```
print()#blank line
print('Get and display keys')
keys = dumbDb.keys()
for key in keys:
    print(key)

print()
print('Attempt to pop and display two values')
print(dumbDb.pop(b'phone','no phone'))
print(dumbDb.pop(b'weight','no weight'))

print()
print('Get and display keys and values')
print('Database length',len(dumbDb))
items = dumbDb.items()
for item in items:
    print(item)

print()
print('Delete a key/value pair and display again')
del dumbDb[b'cubicle']
print('Database length',len(dumbDb))
items = dumbDb.items()
for item in items:
    print(item)

dumbDb.close()
```

```
================================================================================
===
Get and display values
b'name'
b'age'
b'phone'
b'marital'
b'cubicle'
b'gender'

Get and display keys
b'Joe'
b'39'
b'333.444.5555'
b'married'
b'639'
b'male'

Attempt to pop and display two values
b'333.444.5555'
no weight

Get and display keys and values
Database length 5
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'cubicle', b'639')
(b'gender', b'male')
```

```
Delete a key/value pair and display again
Database length 4
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'gender', b'male')
```

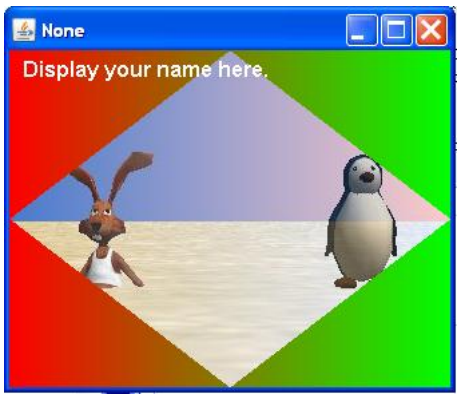
Go to answer 15

Figure index

- Figure 1 . Question 11.
- Figure 2 . Answer 11.
- Figure 3 . Question 12.
- Figure 4 . Question 13.
- Figure 5 . Answer 13.
- Figure 6 . Question 14.
- Figure 7 . Question 15.
- Figure 8 . Answer 15.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.

Answers

Answer 15

False. The output is shown in Figure 8 . The **keys** and the **values** were switched in the bottom panel of Figure 7 .

Figure 8 . Answer 15.

```
Get and display values
b'Joe'
b'39'
b'333.444.5555'
b'married'
b'639'
b'male'

Get and display keys
b'name'
b'age'
b'phone'
b'marital'
b'cubicle'
b'gender'

Attempt to pop and display two values
b'333.444.5555'
no weight

Get and display keys and values
Database length 5
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'cubicle', b'639')
(b'gender', b'male')

Delete a key/value pair and display again
Database length 4
(b'name', b'Joe')
(b'age', b'39')
(b'marital', b'married')
(b'gender', b'male')
```

Go back to Question 15

Answer 14

True.

Go back to Question 14

Answer 13

False. The output is shown in Figure 5 . The **in** operator searches for a match in the **keys** and does not search for a match in the **values** .

Figure 5 . Answer 13.

```
Not in dumbDb
```

Go back to Question 13

Answer 12

True.

Go back to Question 12

Answer 11

False. The code in Figure 1 produces an error by attempting to store a number as a value. The output *(with some text deleted for brevity)* is shown in Figure 2 .

Figure 2 . Answer 11.

```
Traceback (most recent call last):
  File "1359-1930-11.py", line 11, in <module>
    dumbDb[b'aNumber'] = 1234
...
    raise TypeError("values must be bytes or strings")
TypeError: values must be bytes or strings
```

Go back to Question 11

Answer 10

True. See The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases" .

Go back to Question 10

Answer 9

True as of 11/09/2014.

Go back to Question 9

Answer 8

True. See comments in sample code at The Python Standard Library - 12.5. dbm - Interfaces to Unix "databases" .

Go back to Question 8

Answer 7

False. The object returned by **dbm.open** supports the same basic functionality as ***dictionaries*** . Keys and their corresponding values can be stored, retrieved, and deleted. The **in** operator and the **keys** method are available, as well as **get** and **setdefault** .

Go back to Question 7

Answer 6

False. The function named **dbm.open** can be called to open an existing **dbm** database file or to create a new **dbm** database file.

Go back to Question 6

Answer 5

True.

Go back to Question 5

Answer 4

False. **dbm** is a generic interface to variants of the DBM databases - **dbm.gnu** or **dbm.ndbm** . If neither of these modules is installed, the slow-but-simple implementation in module **dbm.dumb** will be used.

Go back to Question 4

Answer 3

False. As of 11/09/2014, no such warning appears on the referenced webpage. Therefore, it might be concluded that even though **shelve** has features that are not available in **dbm** , **dbm** is the more secure of the two. As a result, this course will concentrate on **dbm** and ignore **shelve** .

Go back to Question 3

Answer 2

True as of 11/09/2014.

Go back to Question 2

Answer 1

True as of 11/09/2014.

Go back to Question 1

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-1930-Dbm and Shelve Databases
- File: Itse1359-1930.htm
- Published: 11/10/14
- Revised: 07/28/16

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2110-Preface to Putting Python to Work
Preface to the sub-collection titled Putting Python to Work

Table of contents

- Discussion
- Miscellaneous

Discussion

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX.

This material is provided for the benefit of those students who are interested in expanding their learning experience beyond the minimum requirements of the course titled *ITSE 1359 Introduction to Scripting Languages: Python* . Knowledge of the material in this sub-collection titled **Putting Python to Work** is not a requirement of the course.

During my 20+ years of teaching computer programming, I have learned that many students are more likely to succeed in the course if I provide interesting and useful projects for them to work on in addition to simply computing sales-tax tables, etc. Therefore, I try to provide projects such as image processing, network programming, event-driven programming, game programming, etc. in addition to the more boring text-only types of projects in most of the courses that I teach.

Several interesting libraries and Ebooks for doing this sort of programming with Python are freely available on the web including the following:

- Pygame - special effects, simulations, and games.
- Blender - 3D modeling
- Guzdial's book - media computation (images, audio, etc.)
- Python at Codecademy - an interactive Python tutorial
- Livewires - computer animation and programming
- Matplotlib - 2D plotting

- Python Imaging Library (PIL) - image processing
- Pillow - image processing
- John Zelle's Graphics Module - graphics and animation
- Pythonxy - scientific and engineering software for numerical computations
- Numpy and SciPy - mathematical and numerical routines in pre-compiled, fast functions often required by other libraries.
- The Invent with Python Bookshelf - A large number of free downloadable Python Ebooks

Many of the items in the above list could be used to develop interesting and useful programming projects for use in this course. There is a problem, however. Many of the items in the list were developed using Python version 2.x and have not yet been updated for compatibility with Python 3.x or later as of January 2015. Therefore, some of the items in the above list are not compatible with the version of Python being used for this course.

As time goes on, I plan to monitor the state of these items and to update the contents of this sub-collection to explore the use of the items in this portion of the course material. Therefore, you should consider this portion of the course material to be a work in process.

I will begin with Pygame , which is compatible with Python version 3.4.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2110-Preface to Putting Python to Work
- File: Itse1359-2110.htm
- Published: 01/02/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer
This module introduces the code visualizer provided by the Online Python Tutor.

Table of contents

- Preface
- Discussion
    - Stepping through the program
    - A step-through session
    - Sharing a step-through session
- Miscellaneous

Preface

This module is one in a collection of modules on Python designed for teaching *ITSE 1359 Introduction to Scripting Languages: Python* at Austin Community College in Austin, TX. This module concentrates on the use of the Online Python Tutor , which includes a free educational code visualizer tool created by Philip Guo .

This code visualizer tool helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this code visualizer tool, a teacher or student can write Python, Java, and JavaScript programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs.

Discussion

The primary benefit of this code visualizer tool is that it allows a student to step through a program one instruction at a time, using Python 2.7 or Python 3.3, and to see what the computer is doing as the program executes.

A secondary benefit is the ability to share a step-through session with another person online and to discuss the behavior of the program while stepping through the program.

Stepping through the program

You have to experience this code visualizer tool in order to really appreciate it. Rather than trying to explain it myself, I am going to refer you to a web resource that incorporates the code visualizer tool. The interactive online book titled How To Think Like a Computer Scientist uses this code visualizer tool to create a feature known in the book as **CodeLens** . According to the book,

*"...you can also execute Python code with the assistance of a unique visualization tool. This tool, known as **codelens** , allows you to control the step by step execution of a program. It also lets you see the values of all variables as they are created and modified."*

The chapter titled The while Statement in that book contains an excellent example of stepping through a program using the code visualizer tool. I will refer you to that example to gain an appreciation of the code visualizer tool.

A step-through session

In addition to using the code visualizer tool after it has been incorporated into interactive documents such as the book discussed above, students can also use the code visualizer tool directly . In this case, you simply

- Select your version of Python from a drop-down list.
- Type or paste your code into a text window.
- Press a button labeled Visualize Execution.

That causes four more buttons with the following labels to appear:

- First

- Back
- Forward
- Last

You can click those buttons to navigate forward or backward through the program while observing the information that is displayed on the screen to the right of the code box.

Links to several Python example programs are provided near the bottom of the page here .

Sharing a step-through session

There is a button in the upper-left corner of this page labeled **Start a shared session** . The label on the button describes its purpose.

Right above the button, there is a hyperlink that reads What are shared sessions? Selecting that link will start a video describing how you can share and discuss a session with a tutor or other individual on another computer somewhere on the Internet. The sharing interface includes a text chat box for discussing the session. If you can establish an audio or video connection *(using something like Google Hangout)* you can also discuss the behavior of the program by voice while stepping through and jointly observing the behavior of the program.

All of this capability is available free of charge, but the author does request that you provide feedback to help in his research effort.

Run the program

I encourage you to go online and run some of the example programs that are provided near the bottom of the page here . I also encourage you to open the online book titled How To Think Like a Computer Scientist and experiment with the **codelens** feature of that book.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Itse1359-2510-Getting Started with the Online Python Tutor Code Visualizer
- File: Itse1359-2510.htm
- Published: 02/15/15

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-